REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-02-

0069

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		11-01-1997 - 10-31-2001

4. TITLE AND SUBTITLE
Specification and Dynamic Checking of Composition Constraints In Distributed Component-Based Systems

5. FUNDING NUMBERS
F49620-98-1-0061

6. AUTHOR(S)
van der Hoek, Adriaan W.
Rosenblum, David S.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California, Irvine
115 Administration
Irvine, CA 92697-1875

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
801 N. Randolph Street Room 732
Arlington, VA 22203-1977

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-98-1-0061

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

13. ABSTRACT *(Maximum 200 words)*

Component-based software engineering has been a dream for at least 30 years, beginning with Doug McIlroy's seminal presentation at the 1968 NATO Conference in Garmisch) The dream is rapidly becoming a reality with the advent of component interoperability standards such as ActiveX and JavaBeans, and middleware infrastructures such as NET, DCOM, and CORBA. Both civilian and military software development efforts stand to reap enormous benefits from this technology, in terms of reduced time-to-deployment, reduced development costs, increased productivity, and increased tolerance for complexity. While existing component technologies provide the basic building blocks for a component-based style of development, they still lack the fundamental mechanisms needed to ensure that systems are composed in a manner that ensures the integrity of component interactions. This research has been dedicated to investigating such fundamental mechanisms, In particular, the research has created mechanisms for specifying and checking component compositions in distributed component-based software systems. The research was conducted along two avenues: developing architectural foundations for developing component-based software; and exploiting and extending component standards to support constraint checking. The results of these two avenues of research are described further below. We first briefly discuss each of the two research avenues and the projects that have resulted from pursuing these avenues. Then, we list the significant results achieved by each of the project. The publications that have been produced by this research are listed fully in Section 4

14. SUBJECT TERMS			15. NUMBER OF PAGES 194
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Specification and Dynamic Checking of Composition Constraints in Distributed Component-Based Systems

Final Report

Adriaan W. van der Hoek, Principal Investigator (01-04-2001 – 10-31-2001)
David S. Rosenblum, Principal Investigator (11-01-1997 – 01-04-2001)
Department of Information and Computer Science
444 Computer Science
University of California, Irvine
Irvine, CA 92697-3425

andre@ics.uci.edu
http://www.ics.uci.edu/~andre
+1 949 824-6326 (voice)
+1 949 824-1715 (fax)

For the period 1 November 1997 through 31 October 2001
Grant number F49620-98-1-0061

Prepared for The Air Force Office of Scientific Research

1 Introduction

Component-based software engineering has been a dream for at least 30 years, beginning with Doug McIlroy's seminal presentation at the 1968 NATO Conference in Garmisch.[1] The dream is rapidly becoming a reality with the advent of component interoperability standards such as ActiveX and JavaBeans, and middleware infrastructures such as .NET, DCOM, and CORBA. Both civilian and military software development efforts stand to reap enormous benefits from this technology, in terms of reduced time-to-deployment, reduced development costs, increased productivity, and increased tolerance for complexity.

While existing component technologies provide the basic building blocks for a component-based style of development, they still lack the fundamental mechanisms needed to ensure that systems are composed in a manner that ensures the integrity of component interactions. This research has been dedicated to investigating such fundamental mechanisms. In particular, the research has created mechanisms for specifying and checking component compositions in distributed component-based software systems. The research was conducted along two avenues:

- developing architectural foundations for developing component-based software; and
- exploiting and extending component standards to support constraint checking.

The results of these two avenues of research are described further below. We first briefly discuss each of the two research avenues and the projects that have resulted from pursuing these avenues. Then, we list the significant results achieved by each of the project. The publications that have been produced by this research are listed fully in Section 4.

2 Research Directions

2.1 Developing architectural foundations for distributed component-based software

Software architectures are software system models that represent a software system design at a high level of abstraction. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. Details at lower levels of abstraction, such as the selection of data structures and algorithms for individual modules, are the concern of later stages of design. Much of the research we have conducted used C2 as the architectural vehicle. The C2 architectural style, which has been a focus of study at UC Irvine for several years, is primarily concerned with high-level system composition issues, rather than particular behavioral or component packaging issues. This work has involved four projects:

(1a) Developing an approach to support heterogeneous typing for modeling and evolution of architectural models. Object-oriented subtyping provides a natural way of modeling and evolving component definitions and interactions, and of specifying component behaviors in a way that

[1] See M.D. McIlroy, "Mass Produced Software Components", pp. 88–98 of P. Naur, B. Randell, and J.N. Buxton, *Software Engineering: Concepts and Techniques: Proceedings of the conferences sponsored by the NATO Science Committee, held at Garmisch, Germany, Oct. 7-11, 1968 and Rome, Italy, Oct. 27-31, 1969*, New York: Petrocelli/Charter, 1976.

facilitates checking architectural integrity. Our experience building applications in the C2 style has demonstrated that no one subtyping system is sufficient to support specification and checking of evolvable component compositions. Instead, multiple subtyping systems must be simultaneously supported at the architectural level. For instance, strictly monotone subclassing allows incremental extension of a component's behavior within an architecture in a way that preserves previously established architectural properties. In contrast, implementation conformance with multiple interfaces provides a systematic way of using the same component implementation within different architectural contexts (by 'wrapping' the component with context-dependent interfaces).

(1b) Modeling and exploiting middleware-induced architectural styles. Our studies from Year 1 of the grant demonstrated many inadequacies of existing ADLs with respect to their support for later stages of design and implementation. Additional studies conducted with Elisabetta Di Nitto of CEFRIEL/Politecnico di Milano and Alfonso Fuggetta demonstrate further inadequacies with respect to preserving architectural properties in implementations. These latter studies focus on the notion of a middleware-induced architectural style, which reflects how middleware infrastructures impose constraints on architectures and how architectural models constrain middleware choices. The studies are directed towards investigating how middleware constraints and choices can be reflected in architectural models.

(1c) Supporting architectural modeling in the Unified Modeling Language (UML). To bring architectural modeling into the mainstream of software development, standard design notations must be extended to support modeling of architectural concerns. These extensions must be achieved in such a way that they seamlessly work with and exploit the capabilities of existing tools for the notations. We have developed techniques for extending the Unified Modeling Language (UML) to support architectural modeling. UML is an object-oriented design notation that is rapidly becoming the de facto standard notation of choice for many software development organizations. UML provides extensibility mechanisms in the form of tagged values (valued properties that can be attached to UML model elements), constraints (formal logical expressions constraining the use of UML model elements) and stereotypes (named collections of related tagged values and constraints that essentially provide a new building block for models). These extensibility mechanisms allow us to incorporate support for architectural modeling in UML in a way that preserves the resulting notation's compatibility with existing UML tools.

(1d) Exploring foundations for event-based interoperability across wide-area networks. Asynchronous event notification is a natural interaction paradigm for many classes of distributed component-based applications. Many technologies exist to support event-based interaction in local-area networks, but these technologies are unable to scale to the demands of applications deployed across wide-area networks such as the Internet, where the numbers of components and events may be enormous, and where component interactions span trust domains. In the Siena project, we have explored event data models, event notification service architectures, and routing algorithms to support asynchronous event-based interaction in wide-area distributed component-based applications. Siena supports an advertise/publish/subscribe style of interaction between components and the Siena service. In particular, Siena clients advertise to Siena the classes of event notifications that they publish, they publish those notifications to Siena, and they subscribe with Siena for patterns of notifications. It is the job of Siena to match notifications with subscriptions, and to exploit advertisements and subscriptions in an effort to maximize the scalability of the whole Siena service.

2.2 Exploiting and extending component standards to support constraint checking in component-based software

There is a great deal of interest in emerging component standards such as ActiveX, CORBA and JavaBeans. The emphasis of these standards is primarily on syntactic interface description and introspection, component packaging, runtime binding and wrapping. Our research is based on the Java programming language and the JavaBeans component interoperability model because of their widespread popularity, the wide availability and low cost of their support tools, and the flexibility and extensibility offered by their designs.

We have created extensions to the JavaBeans design pattern to support specification and checking of bean compositions. These extensions are embodied in enhanced versions of the Sun Beans Development Kit (BDK), a free JavaBeans development environment provided by Sun Microsystems in source form. In particular, we have studied two important classes of compositional properties that can be checked in the style of beanbox composition—architectural style constraints and component behavior constraints.

3 Summary of Significant Results

(1a) We have developed a formal theory of heterogeneous typing that allows components in architectural models to be evolved in a systematic way according to different subtyping mechanisms. In particular, the formal model supports analysis and checking of an architecture model containing formally defined interfaces, and it supports incremental evolution of the architecture via subclassing of interfaces. The theory supports particular formulations of naming, interface, behavioral, and implementation subtype conformance, but it can easily be extended to support other formulations.

We have created an ADL called C2SADEL to support architectural modeling in the style needed for specification and checking of component compositions according to our theory of heterogeneous subtyping. C2SADEL is supported by an environment called DRADEL that parses, style-checks and type-checks architectural models expressed in C2SADEL and generates implementation code templates that partially preserve the specified architectural properties.

We have continued our work with C2SADEL and DRADEL by integrating them with an environment called ArchStudio 2.0, which embodies contributions from a number of projects in the software group at UC Irvine, as well as some commercial off-the-shelf tools. ArchStudio 2.0 provides lifecycle-wide support for architecture-based development, and the capabilities of DRADEL form the centerpiece of the environment, allowing other tools to base their work on the architecture models constructed with DRADEL. For instance, the tool ArchShell provides support for runtime modification of a system, where the modifications are performed in terms of the models produced with DRADEL. Other tools in the environment exploit C2SADEL models for later stages of design, for implementation construction and debugging, and for testing.

(1b) Our initial work has focused on middleware for distributed event-based systems, and it produced an evaluation of existing ADLs as to their suitability for modeling and exploiting middleware-induced architectural styles. Additional work has involved characterizing middleware-induced styles in terms of fundamental properties such as control logic (push versus pull) and synchronization style (synchronous versus asynchronous), and in terms of compatibilities between differing properties. Such characterizations will eventually allow us to

identify inconsistencies between the style induced by a middleware infrastructure and the architectural properties of an application whose implementation is to be based on the middleware.

(1c) Our UML extensions (in the form of stereotypes) now support three kinds of architectural modeling. One set of extensions supports creation of UML designs whose classes and inter-class relationships (as expressed in UML class diagrams) adhere to the rules of the C2 architectural style. Another set of extensions supports modeling and analysis of UML class behaviors (as expressed in UML state transition diagrams) according to the precepts of the ADL Wright. The third set of extensions supports modeling and analysis of UML class behaviors (as expressed in UML state transition diagrams) according to the precepts of the ADL Rapide. In particular, the Rapide extensions capture Rapide's model of partially ordered sets of events (event posets) and event pattern constraints over the poset produced by an architecture.

The experience with Rapide has begun to reveal limitations in our ability to support architectural modeling in UML. In particular, we discovered weaknesses and ambiguities in the semantics of UML's event model that prevent a full realization of Rapide's event model in UML.

(1d) We have investigated three classes of distributed server architectures for Siena--a hierarchical architecture (which embodies an asymmetric master/slave dependency between servers in different hierarchical levels), an acyclic peer-to-peer architecture (in which there is a single path between any pair of servers), and a general peer-to-peer architecture (in which there may be multiple paths between servers). For each architecture, we have defined routing protocols that efficiently route service requests between Siena clients. We have performed a wide range of simulation studies that characterize the scalability of each class of architectures in terms of data curves of message costs, message numbers, and other salient characteristics.

For the routing algorithms, we have defined a number of optimizations that increase the efficient and scalability of the event notification services provided by Siena. Siena uses a form of routing based on notification content to establish interaction paths between advertisers, publishers and subscribers. Siena employs the principle of downstream replication, whereby notifications that match multiple subscriptions are routed as far as possible in a single copy before they are replicated for delivery to the multiple subscriber, in order to minimize the number of notification copies being routed. Siena employs the principle of upstream evaluation, whereby subscriptions are matched against notifications are matched as close as possible to the source of notifications, in order to limit the routing of unsubscribed notifications. And Siena is able to combine subscriptions that match overlapping sets of notifications into more general notifications in order to limit the number of subscriptions being routed. We have developed a formal model of content-based routing that synthesizes many of the insights we have gleaned from these optimizations.

(2) We have created a tool called ARABICA that checks the rules of the C2 architectural style as an architect incrementally creates compositions. ARABICA provides special C2 component and connector beans, and it also can wrap off-the-shelf beans to behave as C2 components. ARABICA exploits previously developed approaches to wrapping off-the-shelf components for use in C2-style architectures. To wrap an off-the-shelf bean, ARABICA interactively queries the architect to identify which bean events behave as C2 requests and which behave as C2 notifications. Working with ARABICA has revealed weaknesses in this wrapping approach. In particular, components that contain internal references to other components (or references to other components in method signatures) may violate architectural abstractions in ways that are

hidden to, or difficult to detect by, an architecture-based composition environment such as ARABICA. We have achieved an initial integration of ARABICA with the ArchStudio 2.0 mentioned above.

We have also created a tool called ROBUSTA that recognizes an extended version of the JavaBeans design pattern, whereby a component's behavior is specified through pre- and post-conditions on the component's methods and invariants on the component class. In the spirit of the JavaBeans design pattern, these constraints are provided as additional methods of the component, which are named according to a particular syntactic pattern expected by ROBUSTA. ROBUSTA detects the additional constraint-checking methods through introspection and establishes calls to them as components are incrementally composed via the JavaBeans visual style of component composition (using drag-and-drop connection of event-generating beans to event handler methods in event-listening beans). In addition, ROBUSTA generates calls to constraints associated with bean property methods, so that changes to property values are checked for consistency with the associated constraints. In all cases the calls are established using assertion-checking techniques previously created by the PI for C and Ada. When the composed application is executed, the constraint-checking methods are called at appropriate points in the execution.

4 Publications

4.1 Accepted Peer-Reviewed Publications

[CDR+98] A. Carzaniga, E. Di Nitto, D.S. Rosenblum and A.L. Wolf, *Issues in Supporting Event-based Architectural Styles*, **Proceedings of the Third International Software Architecture Workshop** (ISAW-3), Lake Buena Vista, Florida, Nov. 1998, pp. 17–20.

[CRW00] A. Carzaniga, D.S. Rosenblum and A.L. Wolf, *Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service,* **Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing** *(PODC 2000)*, Portland, Oregon, July 2000, pp. 219–227.

[DR99] E. Di Nitto and D.S. Rosenblum, *Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures,* **Proceedings of the 21st International Conference on Software Engineering** (ICSE '99), Los Angeles, May 1999, pp. 13–22.

[LR01] C. Lüer and D.S. Rosenblum, *Wren—An Environment for Component-Based Development,* **Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering** (ESEC/FSE '01), Vienna, Austria, Sept. 2001, pp. 207–217.

[LRH01] C. Lüer, D.S. Rosenblum and A. van der Hoek, *The Evolution of Software Evolvability,* **International Workshop on Principles of Software Evolution** (IWPSE '01), Vienna, Austria, Sept. 2001, pp. 127–130.

[MR99] N. Medvidovic and D.S. Rosenblum, *Assessing the Suitability of a Standard Design Method for Modeling Software Architectures*, **Proceedings of the First Working IFIP Conference on Software Architecture** (WICSA1), San Antonio, Texas, Feb. 1999, pp. 161–182.

[MRR+] N. Medvidovic, D.S. Rosenblum, J.E. Robbins and D.F. Redmiles, *Modeling Software Architectures in the Unified Modeling Language,* **ACM Transactions on Software Engineering and Methodology,** to appear.

[MRT99] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, *A Language and Environment for Architecture-Based Software Development and Evolution*, **Proceedings of the 21st International Conference on Software Engineering** (ICSE '99), Los Angeles, California, May 1999, pp. 44–53.

[MTR98] N. Medvidovic, R.N. Taylor and D.S. Rosenblum, *An Architecture-Based Approach to Software Evolution*, **Proc. ICSE '98 International Workshop on the Principles of Software Evolution**, Kyoto, Japan, Apr. 1998, pp. 11–15.

[NR98a] R. Natarajan and D.S. Rosenblum, *Merging Component Models and Architectural Styles*, **Proceedings of the Third International Software Architecture Workshop** (ISAW-3), Lake Buena Vista, Florida, Nov. 1998, pp. 109–111.

[OHR00] A. Orso, M.J. Harrold and D.S. Rosenblum, *Component Metadata for Software Engineering Tasks,* **Proceedings of the 2nd International Workshop on Engineering Distributed Objects** (EDO '00), Davis, California, Nov. 2000, pp. 126–140.

[OTG+99] P. Oreizy, R.N. Taylor, M.M. Gorlick, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum and A.L. Wolf, *An Architecture-based Approach to Self-Adaptive Software*, **IEEE Intelligent Systems,** vol. 14, 3, 1999, pp. 54–62.

[RMR+98] J.E. Robbins, N. Medvidovic, D.F. Redmiles and D.S. Rosenblum, *Integrating Architecture Description Languages with a Standard Design Method*, **Proc. 20th Int'l Conf. on Software Engineering**, Kyoto, Japan, Apr. 1998, 209–218.

[RN00] D.S. Rosenblum and R. Natarajan, *Supporting Architectural Concerns in Component Interoperability Standards,* **IEE Proceedings—Software,** vol. 147, 6, 2000, pp. 215–223.

[Ros98] D.S. Rosenblum, *Challenges in Exploiting Architectural Models for Software Testing*, invited submission, **Proc. NSF/CNR Workshop on the Role of Software Architecture in Testing and Analysis**, Marsala, Sicily, Jul. 1998, pp. 49–53.

[RRR97] J.E. Robbins, D.F. Redmiles and D.S. Rosenblum, *Integrating C2 with the Unified Modeling Language,* **Proc. 1997 California Software Symposium**, Irvine, CA, Nov. 1997, pp. 11–18.

[RWC98] D.S. Rosenblum, A.L. Wolf and A. Carzaniga, *Critical Considerations and Designs for Internet-Scale, Event-Based Compositional Architectures*, **Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures**, Monterey, CA, Jan. 1998, 4 pp.

4.2 Submitted Peer-Reviewed Publications

[MRT] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, *Heterogeneous Typing for Software Architectures,* submitted to **ACM Transactions on Software Engineering and Methodology.**

4.3 Technical Reports

[CRW98] A. Carzaniga, D.S. Rosenblum and A.L. Wolf, *Design of a Scalable Event Notification Service: Interface and Architecture,* Department of Computer Science, University of Colorado at Boulder, Technical Report CU-CS-863-98, 1998.

[LR00] C. Lüer and D.S. Rosenblum, *Wren—An Environment for Component-Based Development,* Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-00-28, 2000.

[MRT98] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, *A Type Theory for Software Architectures,* Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-98-14, 1998.

[ORT98] P. Oreizy, D.S. Rosenblum and R.N. Taylor, *On the Role of Connectors in Modeling and Implementing Software Architectures,* Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-98-04, 1998.

[NR98b] R. Natarajan and D.S. Rosenblum, *Extending Component Interoperability Standards to Support Architecture-Based Development,* Department of Information and Computer Science, University of California, Irvine, Technical Report UCI-ICS-98-43, 1998.

from the Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering FSE-9, published as Software Engineering Notes 26, 5 (2001), pages 207-217

WREN—An Environment for Component-Based Development

Chris Lüer David S. Rosenblum

Institute for Software Research and
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
+1-949-824-{2703,6534}

{chl,dsr}@ics.uci.edu

ABSTRACT

Prior research in software environments focused on three important problems—tool integration, artifact management, and process guidance. The context for that research, and hence the orientation of the resulting environments, was a traditional model of development in which an application is developed completely from scratch by a single organization. A notable characteristic of component-based development is its emphasis on integrating independently developed components produced by multiple organizations. Thus, while component-based development can benefit from the capabilities of previous generations of environments, its special nature induces requirements for new capabilities not found in previous environments.

This paper is concerned with the design of *component-based development environments*, or CBDEs. We identify seven important requirements for CBDEs and discuss their rationale, and we describe a prototype environment called WREN that we are building to implement these requirements and to further evaluate and study the role of environment technology in component-based development. Important capabilities of the environment include the ability to locate potential components of interest from component distribution sites, to evaluate the identified components for suitability to an application, to incorporate selected components into application design models, and to physically integrate selected components into the application.

Categories and Subject Descriptors

D.2.6 **[Software Engineering]**: Programming Environments – *Graphical environments, Integrated environments, Interactive environments;* D.2.2 **[Software Engineering]**: Design Tools and Techniques – *Modules and interfaces, Software libraries;* D.2.11 **[Software Engineering]**: Software Architectures – *Information hiding, Languages;* D.2.9 **[Software Engineering]**: Management – *Life cycle, Software configuration management;* D.2.7 **[Software Engineering]**: Distribution, Maintenance and Enhancement; D.2.13 **[Software Engineering]**: Reusable Software – *Reusable libraries, Reuse models;* D.1.5 **[Programming Techniques]**: Object-oriented Programming.

General Terms

Design, Languages.

Keywords

Component-based software engineering, Java, Java Beans, software components, software environments.

1. INTRODUCTION

It has been stated that component technology, while successful in industry, has not received the attention it deserves from the research community [16]. Industrial component models are still rudimentary, and the approaches of different vendors vary strongly. Research is necessary in order to define a common foundation of component technology, and to identify areas in which current standards and tools have to be extended.

Software environments are one area that can benefit especially well from further research. Software development environments (SDEs) were originally designed to integrate collections of tools and to manage locally created development artifacts. Later, process centered software engineering environments (PSEEs) were developed to facilitate the use of well-defined processes to guide development. In order to provide tool integration and process-based guidance for the special needs of component-based development, we envision a new generation of environments, *component-based development environments*, or *CBDEs*. Reusable components developed by and licensed from other organizations cannot be treated in the same way as artifacts that were developed in-house, since it is usually not possible to change or analyze their implementations. Therefore, new approaches are needed to support identification, retrieval and integration of such components within an environment in an Internet-scalable way.

Szyperski defines a component as follows [32]:

- *A unit of independent deployment.* This means that a key goal of component technology is to facilitate code reuse [14]. A component is a piece of code that has been prepared for reuse. This is opposed to code scavenging, where code that

was not explicitly intended to be reusable is being reused. Though initially more expensive, we view design-for-reuse as being the superior approach to enabling reuse.

- *A unit of third-party composition.* Reuse will pay off only when reusing a component that was developed by another organization is significantly easier than redeveloping it. In the ideal case, an application would be composable from components by domain experts without actual programming.
- *Without persistent state.* A component is a piece of code, or a set of abstract data types. In an object-oriented system, a component is a set of classes. A component is not an object or a set of objects.

A CBDE must provide its users with information about components. The users have not designed the components themselves, so they depend on the environment to learn about them. With the use of components, the focus of tools shifts from implementation to design, since the goal of component reuse is to minimize implementation effort. Users must decide which components fit best into their architecture, so the environment should be able to visualize the dependencies among the components. Because components are developed by third parties, the environment should provide the means to access components located at remote sources.

In this paper, we present requirements for CBDEs, and we describe a prototypical environment, WREN, which we are building based on these requirements. Our prototype is based on the Java language and the Java Beans component model. Components packaged as described in this paper are backwards compatible with Java Beans, although they have been extended in various ways. As described in the paper, WREN serves as an early exemplar of this new generation of environments.

2. REQUIREMENTS OF COMPONENT-BASED DEVELOPMENT ENVIRONMENTS

While a number of specialized technologies have been produced in both research and industry to facilitate particular aspects of component programming and reuse, we are unaware of any attempts to provide comprehensive, integrated environment support for the full range of lifecycle activities that must be undertaken in component-based development. In this section we identify seven requirements for CBDEs that address the needs of component-based development. Some of these requirements are addressed by industrial component models, while some of them are not yet widely adopted and are perhaps even controversial. Briefly,

- Accepted rules of *modular design* should be supported explicitly. The environment should support a separation between the private and public parts of a component.
- The environment should support and exploit component *self-description*, meta-information that is stored directly inside of the component. It is used in a limited way in industrial component models like Java Beans and COM.
- Components should be defined and accessed within a *global namespace of interfaces*, which provides a method to name interfaces in a globally (worldwide) unique way. This reduces the problem of semantics matching to one of namespace agreement.
- The environment should support a bipartite development process comprising two parts: *component development* and *application composition*. The former deals more with technical issues of individual component development, while the latter is more application-oriented.
- Application composition consists of configuration of the components and the design and implementation of additional functionality that is not available in components. The environment should support two methods of configuration: *connection* and *adaptation*.
- A CBDE should support *multiple views*, including a development view and a composition view to represent the two halves of the component-oriented process, and a type view and an instance view to show different aspects of the composition view, using an explicit architectural model to represent the overall structure of the application.
- The maintenance problems associated with component technology should be addressed by the environment through *reuse by reference,* which is supported via access to remote, searchable component repositories.

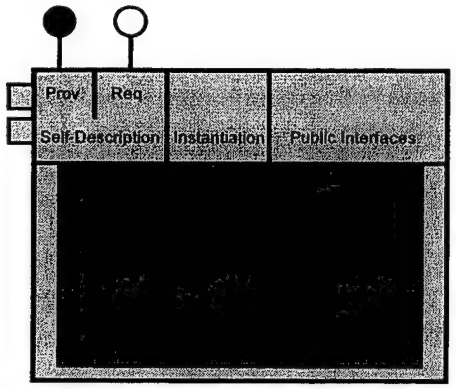


Figure 1. Structure of a Component. Public parts are light gray and private parts are dark gray.

We next discuss the rationale for these requirements.

2.1 Modular Design

Figure 1 presents a generic model of a component that has been prepared for use in a CBDE. A component should be divided into a public part and a private part according to the principle of information hiding or encapsulation [26]. The private part is not accessible from the outside; it contains implementations (in the form of classes) and resources (for example, graphics or help files). The public part contains the self-description of the component, an instantiation mechanism, and optionally public interface definitions. The instantiation mechanism is necessary so that clients can retrieve instances of the data types implemented by the component. To do so, a client specifies only the interface of the data type of which it wants to retrieve a new instance. The decision of which actual class is used to provide this instance is hidden and made by the component itself. Public interface definitions are interfaces that are contained in the component and made accessible to other components, which might want to implement them. The purpose of the self-description and the *provides* and *requires* ports is described below.

The basic unit of syntactical description is the interface. An interface is a named set of operations that describes an abstract data

type. Explicit interfaces make it possible to provide alternative implementations (in the form of classes) for a given data type. Thus, if we ensure that components use only interfaces for their specification, the actual implementations will be encapsulated and exchangeable. Interfaces can be specified independently from the components that implement them so that competing manufacturers can offer components that are interchangeable.

2.2 Self-Description

Self-description is a central idea of component technology. Components should be able to provide information about themselves in a systematic way to a CBDE, and to other components at runtime [25]. Description that is contained in the component itself has many advantages over externally stored description. External description, such as documentation stored in text files, can get lost, often has to be updated manually, and cannot easily be queried by development environments. On the other hand, many forms of self-description can be automatically generated and embedded within the component implementation.

The self-description of a component should contain all the information that is needed to reuse it. This is, first, information about the services that the component provides, and second, information about the services the component requires to work [7]. The information in both of these categories can include syntactic, semantic, quality-of-service, and non-technical descriptions [2].

Providing all this information in the component itself instead of in the form of documentation that is stored elsewhere makes the information available to composition tools. A composition tool can check if two components can be connected without having access to their source code, by querying the self-description. In a similar way, component repositories can leverage component self-description for searching and retrieving components. They can check a user's requirements against the self-description. A component self-description standard could reduce the need for a repository standard, because component repositories could then be very simple when all the information about components is stored where it belongs—in the components.

In a similar way, configuration management can be simplified by the use of self-describing components. Typically, configuration management tools store external information about the dependencies between components. This is necessary when arbitrary files are managed. The task becomes easier, however, when the application is built out of self-describing components. Self-description moves dependency information into the components, where it is encapsulated so that it can easily evolve with the evolution of the component implementation.

2.3 Global Namespace of Interfaces

In an ideal situation, component interfaces would be formally specified, and a CBDE would perform formal reasoning to ensure the semantic compatibility of component implementations with their interfaces. However, such reasoning tools are still not widely available or widely used by practitioners, and most commercial components do not have formally specified interfaces. A global namespace of interfaces partly solves the problem of how a CBDE will ensure consistency between the semantics of a provided component and the semantics required of the component; Zaremski and Wing have studied this problem in the context of *signature matching* [36]. While there may be different interfaces providing the same functionality, in a global namespace of interfaces, two interfaces with the same name are intended to be functionally equivalent. On a fundamental level, this greatly simplifies the problem of matching provided components to required semantics, since the problem is reduced to name equality. Only when components do not match at the interface level is human intervention required: Either they are truly incompatible (i.e., incompatible on a semantic level), or the incompatibility is only syntactic, so that they can be matched by simple manual adaptation (for example by wrapping one of them). Of course, mechanisms are still needed to ensure that a component correctly implements the semantics promised by its interfaces, but this problem already existed alongside the component matching problem.

2.4 Component Development and Application Composition Processes

A component-oriented development process looks different from a traditional one. The process is bipartite: The development of components, and the composition of an application from the components, are separated. Typically, the two process parts will be executed by different organizations, the component manufacturer and the organization that wants to license and reuse the manufactured components. We refer to these organizations as the *component developer* and the *application composer,* respectively.

Component development is a traditional development process since all the usual lifecycle phases are traversed. The main difference is that the end product is not a complete application. This means that the product is comparatively small, which may make development processes suited to small projects preferable.

A CBDE can support traditional component development, but it must excel at supporting application composition, which should focus on the business aspects of an application. In the ideal extreme, all components can be *bought* or otherwise obtained, since the goal of component reuse is to minimize the implementation phase of an application. The application composer must select the right components, connect and adapt them, and identify and build components that might be missing. In the near future, it will not be possible to completely eliminate the implementation phase except for trivial projects, but it can be minimized and simplified using appropriate components and environment capabilities.

The application composition process differs from a traditional process in the requirements phase. In requirements, and even more so in design, the component market must be taken into consideration. Finding components that match arbitrary requirements will be difficult or impossible; instead one is forced to select from prepackaged components with given architectural assumptions. The cost savings gained by component reuse will often make it feasible to adapt requirements and design to the components that are available. Thus, the availability of components must be considered during the whole process [21].

2.5 Connection and Adaptation

Once the decision to reuse a certain component is made, it will have to be configured within a CBDE. Component configuration consists of connection and adaptation. Components have to be connected to each other so that they can cooperate. In the simplest case, the connector is just a link between a given required service and a given provided service. In other words, a connector establishes how a requirement is fulfilled. But connectors can be more complex; it is useful to have them encapsulate functionality that logically belongs within a shared infrastructure (for example,

communication protocols in a distributed system) rather than to either of the two components that are being connected [31] [6].

Adaptation increases the value of components [3]. The more flexible and adaptable a component is, the more often it will be reused. Ideally, a component will provide ways for application composers to adapt it. However, a component manufacturer will not be able to foresee all adaptations that might be necessary. For this reason, there should be means to adapt a component externally without having to interact with it, for example wrapping.

2.6 Multiple Views

2.6.1 Development View and Composition View

CBDEs should aide both the viewpoint of the component developer and the viewpoint of the application composer. Although a component developer will not necessarily compose any application, the application composer will have to develop some components that are specific to the application being built. So, the application composer may have to switch between both roles.

The *component development view* of a CBDE will look very much like a traditional, non-component-oriented environment. But it should provide a way to distinguish the public features of a component from its internal, private features. In many languages this is done through corresponding keywords. A specific graphic design notation that shows the outside (the specification) versus the inside (the implementation) is helpful. Further, the code for instantiation and syntactic self-description can easily be generated from a graphical representation, such as a UML class diagram.

The *application composition view* will be less traditional. Most importantly, it must abstract from the hidden internals of the components. Even if a component was written by the composer, and so its internals are accessible, the internals should be hidden. Since the purpose of component technology is to minimize implementation effort, the composition view will look very much like a design view.

2.6.2 Instance and Type View

The composition view should be divided into two subviews. The *type view* will show the components that are used and their dependencies. The *instance view* will show selected instances of some of the data types provided by the components, and how they are configured.

Instance views are known from commercial development environments (for example, Web Gain Visual Café, or IBM Visual Age). They allow the composer to visually adapt and connect certain objects (instances of classes), such as GUI elements in dialogs, menus and so on. Graphical instance views save implementation effort by providing a way to specify trivial code in a visual manner. Unfortunately, their applicability is limited. There is no way to specify dynamic behavior in them, such as instantiation. Objects that cannot be created at program initialization, but only later, cannot be represented. For this reason, instance diagrams are best suited to show objects that are singletons, such as unique GUI dialogs, or a database. They are less suited for objects that represent business logic or container data structures.

Type views are on the same logical level as UML class diagrams, but instead of classes, components are shown, and instead of associations or inheritance relations, connectors are shown. The purpose of the type view is to show how the components depend on each other, which components are used in the application, which might be exchanged, and what might be missing. The composer must be able to see what each component provides and requires, for example in order to identify requirements that are not yet met.

The type view shows the architecture of the application that is being composed, and serves as a basis for design decisions. For example, once a need is identified, the composer will have to search in a *component market* for components that fulfill this need. Typically, more than one such component will be available. The composer can use the type view to check which of them best fits into the architecture, and then this can be used as a selection criterion together with aspects like quality of service or price.

2.6.3 Explicit Architectural Diagrams

The relationship between software architecture and component-based development is not well understood yet [29]. UML can be used to some degree to model software architectures, but it currently lacks key facilities needed for this [18] [19]. For instance, UML component diagrams cannot be used to show unmet requirements since they provide no syntactic notation for entities that are required to exist but do not.

For this reason, we propose *provides* and *requires* ports as a diagrammatic notation. The concept of ports is known from, among others, the architecture description language Darwin [15], and they are also used in UML for Real-Time [30]. A port is a part of a component that is expected to be linked to another port with a connector, but is not necessarily connected at all times. Each port is either a *requires* port or a *provides* port, and connectors are directed from *requires* to *provides*, so that they can be interpreted as use-relations. A *requires* port that is not connected shows that something is missing—the component is not yet ready to be used. An application composer can keep track of the completeness of the application that is being built by watching the status of the ports.

2.7 Reuse by Reference

Component reuse exacerbates the problem of maintenance. An application that consists of a large number of independently bought components will be much harder to update than a traditional application built by a single organization, since each component will have individual updates from its manufacturer. *Reuse by reference* is a possible solution to this problem.

Reuse by reference means that a single, worldwide master copy of a component is stored in a component repository and referenced over the Internet. Copying is performed by the CBDE only in the form of caching for performance purposes. A permanent connection is established by the CBDE between the client application that uses the component and the repository on which the master copy resides, so that the component can be updated automatically.

Component repositories should adhere to a standard that makes close integration with CBDEs possible. This standard should support not only referencing of components as described above, but also allow CBDEs to search for components in a flexible and general way.

3. THE WREN ENVIRONMENT

WREN is a prototypical implementation of an integrated CBDE that we are building to realize and evaluate the requirements discussed in Section 2. WREN is integrated with Web Gain Visual Café [34], a software development environment. WREN is a client

of one or more component repository servers; we have built such a server, which communicates with WREN through a simple protocol that runs on top of TCP/IP.

In the following, we describe the features of WREN[1], its use for application composition, and how it interacts with the other applications. Support for individual component development is planned, but not yet implemented except as supported in Visual Café.

3.1 Programming Language

We chose Java as the programming language for WREN because it supports component technology and addresses our requirements for CBDEs in multiple ways:

- It supports encapsulation through its access modifiers. Java provides encapsulation on two levels, class and package. Since components can contain more than one class, we use the package-level access modifiers to implement components.
- In Java, signature descriptions can be obtained at runtime through the reflection mechanism of the language. This makes it possible to automatically generate component self-descriptions and simplifies component configuration.
- Java supports interfaces as explicit entities similar to classes. This has the advantage that interfaces and classes can be treated uniformly. A component can provide both classes (i.e. implementations of interfaces) and interfaces.
- Java interfaces reside in a global, worldwide namespace, which is created through the naming convention for package names used in Java: A name should start with the reversed Internet domain name of the manufacturing organization. For example, an interface for abstract data type `foo` developed at the University of California, Irvine, could be named `EDU.uci.foo`.
- Java supports dynamic linking and late binding. This makes it possible to quickly build and evaluate different architectures of a component application.

3.2 WREN Components

WREN requires components to have a specific format and to provide a minimum of self-description. Their architecture is analogous to Figure 1.

A WREN component is a Java archive (jar) file that can contain Java classes, interfaces, and resources. It must contain a self-description class, which provides information about the component. In particular, it has methods that return descriptions of the ports of the component (i.e. which data types are required or provided). The self-description class also has to know which private classes implement the abstract data types that the component provides. Since the data types are identified through interfaces only, other components can receive instances of those data types only through the self-description class (see Section 3.5 below). All classes besides the self-description class should be declared package level private.

WREN components can also contain a diagrammatic representation of themselves, so that they can be represented in component diagrams that are created in the environment (see Section 3.4.3).

3.3 WREN Interfaces

Interfaces are central to the WREN component architecture. A component is characterized by the interfaces of the data types it provides or requires.

WREN interfaces are self-description wrappers around Java interfaces that can provide additional natural language documentation, and additional information about individual operations in the interface. Each operation can have an additional self-description wrapper to provide information about itself. In particular, this wrapper can define post-conditions to assert constraints on the execution of the operation at runtime.

Post-conditions can be a pragmatic solution to the problem of component trust in certain cases [20]. Since components are typically built by another organization and shipped without source code, the application composer needs to trust the component and its developers. Post-conditions can be used efficiently when there is a large difference between the complexity of an operation and the complexity of checking it. An example is prime factorization: while it is complex to find the prime factors of a given integer, it is comparatively easy to check if two numbers are indeed prime factors of another one.

WREN interfaces have version numbers. Versioning is linear; there are no concurrent versions. Higher versions are required to be interface-compatible with all older versions; this means that new versions may add operations, but not remove any. Interface versioning makes component versioning unnecessary, because components are fully specified by the interfaces they provide and require. Thus, two different versions of the same component implementation can simply be treated by WREN as two different components that have the same (or similar) specifications.

3.4 Application Composition Process

Figure 2 summarizes the application composition process that is facilitated by WREN. While the process is not currently enforced in any way, the environment is designed to aid each part of this process. After the requirements are identified, relevant components have to be found. As a first step, repositories should be *searched* in a top-down manner; once the most important components are identified, it will be easier to formulate search criteria for the rest. A typical search will produce far more candidates than needed, many of which will be mismatches. So, in the next step, the composer has to *select* among the found components. All levels of component self-description will be used in this activity. Components that have been selected next need to be *configured* (connected and adapted). Now, missing components, which are required by the selected components, have to be found and integrated, so the process loops back to the search step. Unlike the beginning of the process, where components can be searched for only by vague, natural language criteria, the interfaces specified by the *requires* ports can now be used to automatically search for compatible components. There will still be multiple matches, so that the composer will have to select again according to soft criteria such as quality of service. After several iterations, all components that can be reused will have been found and configured.

[1] Sir Christopher Wren (1632–1723) is remembered for his designs of 51 churches rebuilt in London after the Great Fire of 1666. Each design was unique but was a recognizable variant of an elegant new architectural style.

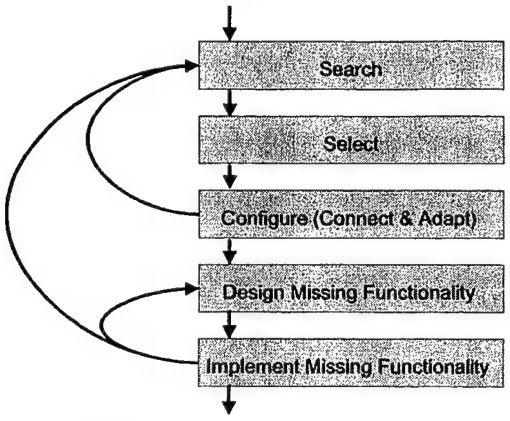


Figure 2. Application Composition Process.

Missing functionality for which no components can be found will have to be *designed* and *implemented* in a traditional manner.

In summary, application composition is an iterative process involving searching, selecting, and configuring components. Searching can be automated in part, but selection and configuration are creative tasks that require design experience.

As shown in Figure 3, WREN supports these activities through repository search and selection views. As a result of the three steps, there are three sets of components that exist during the process. First, there are *available components*, which are all components that match the current search criteria. Out of these, the composer has to select those that are to be used, the *selected components*. Given the set of selected components, the environment can identify *missing components*. These are all the implementations that are required by one of the selected components but not fulfilled by another one. Missing components can only be described in the form of incomplete requirements, since they are not found yet.

3.4.1 Searching for Components

Typically, the application composer will start with a broad search using natural-language keywords. The composer enters the search terms into the CBDE, which in turn sends a search command to all the repository servers it knows about.

Search commands are implemented as pieces of mobile code. A repository server executes the mobile code and allows it to search through all its stored components. The mobile code then queries the self-description of the identified components in order to check them against some associated search criteria. The default search command just checks the search terms against a list of keywords provided by the semantic self-description of a component. However, the repository architecture leaves the decision of how to search to the client CBDEs. A CBDE could easily replace this basic search strategy with a more complex one, for example one that makes use of natural language processing features. The use of mobile code for searching the repository makes the repository itself an almost trivial piece of software. All the management of meta-information, dependencies, and so on that is typically done by a reuse repository is delegated to the components themselves, or rather their self-description.

When a component is found that matches the search criteria, a part of the component is transferred to the client. This part contains the self-description information and can handle calls to the implementation part of the component. WREN adds the component to its set of available components (shown in Figure 3), and uses the component self-description to present information about the component to the composer.

3.4.2 Component Selection

Often, the set of available components will be very large, since it is difficult to specify search criteria in a sufficiently precise way. The application composer uses the Available Components View to browse through the available components, to look at their properties, and to select the ones that are needed (via check boxes).

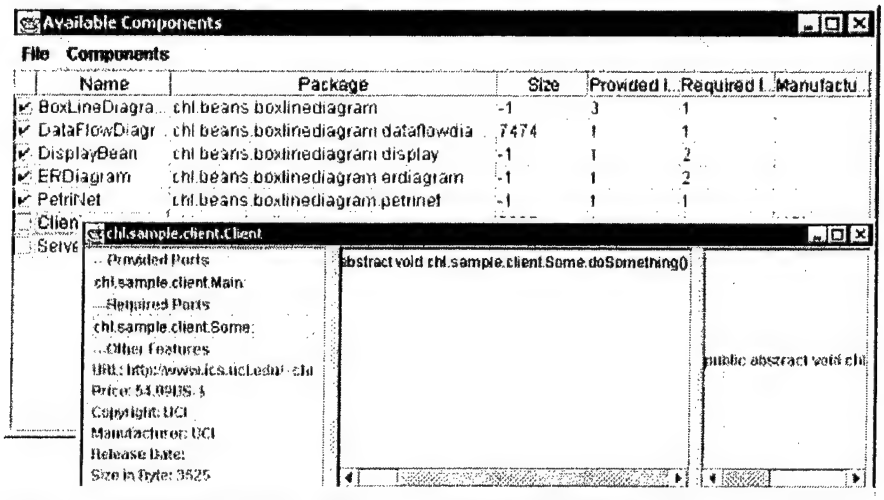


Figure 3. Available Components View and Component Information View. The Available Component View shows if a component is selected, its name, its package name, and selected properties of the component, among them the number of provides ports and of requires ports. The Component Information Window shows all essential information about one component; on the left side general information, in the middle and on the right side the information about the port that is selected.

As also shown in Figure 3, WREN has a window that displays a selection of relevant properties of the available components for easy comparison. Among them are name, manufacturer, size, price, and number of *provides* and *requires* ports. The numbers of ports allow an easy estimation of the architectural complexity of the component. For example, a component that has zero *requires* ports will be at the bottom of the architecture because it does not depend on any other components. An alternate view of the available components is sorted by the interfaces that the components implement, making it easy to compare all components that are possible suppliers for a given data type. However, since a component usually implements more than one interface, this view is less compact.

From the requirements of the selected components, WREN identifies the set of missing components. In particular, it checks through the *requires* ports and adds an entry to the set of missing components for each required data type that is not provided by any of the selected components. It may be possible that several of the missing data types are implemented by one component, so the size of this set does not permit conclusions about the number of actual components that have to be found.

Now, the "find missing components" feature of the environment can be used to automatically search the repositories for all matching components. It is possible that more than one component matches a requirement for a "missing component", so that the composer will have to select among them. The process of searching and selecting components has to be repeated until the set of missing components is empty or the composer decides to reimplement the missing components. To do so, a missing component can be marked as "self-implemented"; this will exclude it from further searches.

3.4.3 Type-Oriented Component Configuration

As shown in Figure 4, WREN has a design editor that allows the composer to connect components. The editor is based on Argo/UML [28] [1], an open-source design environment, and it displays UML component diagrams that are augmented by ports as discussed in Section 2. Components selected from a repository are represented in these diagrams by icons provided in the self-description of the components. When the diagram is opened, all selected components are displayed with their respective *requires* and *provides* ports. *Requires* ports are depicted as hollow circles, *provides* ports as filled circles. Each port is labeled with the name of the interface for which an implementation is required or provided. The composer can drag the components and create directed connections in the form of UML dependencies from *requires* ports to matching *provides* ports. Each *provides* port can be used by any number of *requires* ports, but a *requires* port cannot be connected to more than one *provides* port. It is not possible to change the number or names of the ports of a component, since this would require access to its source code. It should be noted that component adaptation as described in Section 2 is not yet implemented.

A component diagram in this style gives an overview of the architecture that is being built and makes it easy to see which requirements are not yet fulfilled. Each unfulfilled requirement corresponds to a *requires* port that is not connected to any *provides* port. Figure 4 provides an example of this with `DisplayBean`'s *requires* port `Printer`. In a similar way, one can see which components may be affected when a component is exchanged for a compatible one.

To conclude type configuration, the composer must specify the main method of the application, i.e. the method with which execution is started. The environment presents a list of all public methods that could be used as main methods, and the composer chooses one of them.

3.4.4 Instance-Oriented Configuration and Component Deployment

WREN uses Visual Café for instance-oriented configuration. Visual Café is a commercial Java development environment that supports visual connection and adaptation of Java Beans on an instance basis. When the type-oriented configuration described above is completed, the composer can export the components to

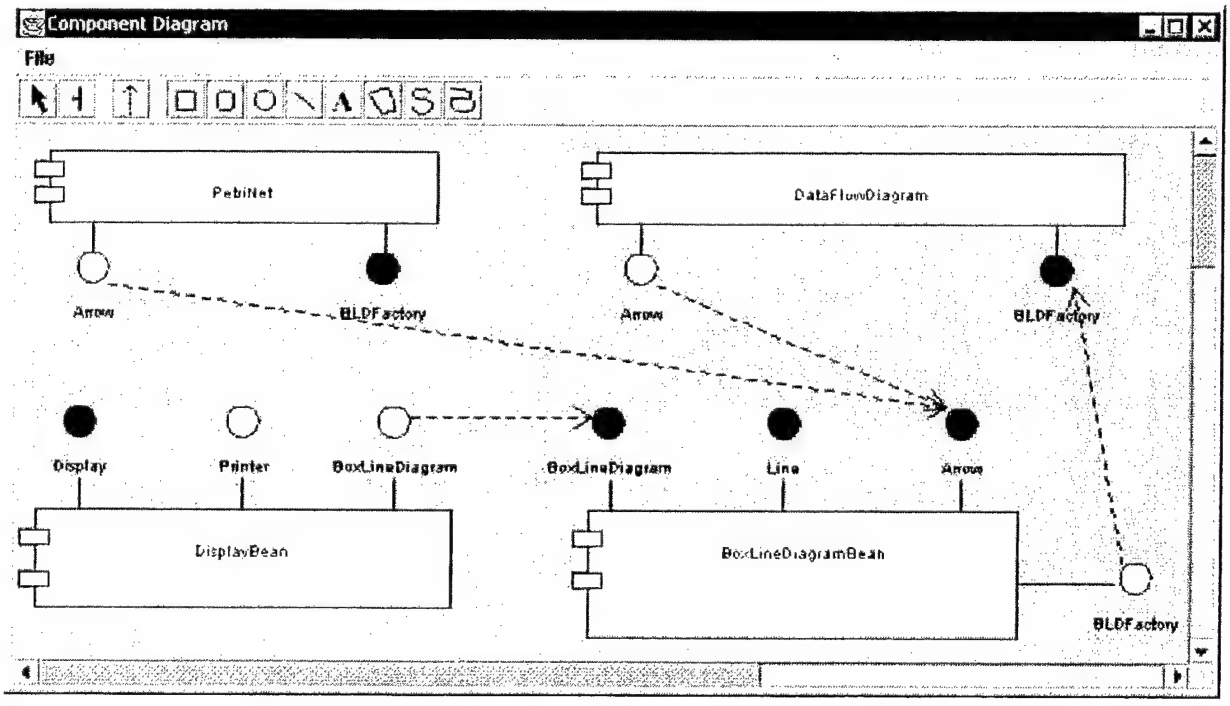


Figure 4. Type-Oriented Component Configuration.

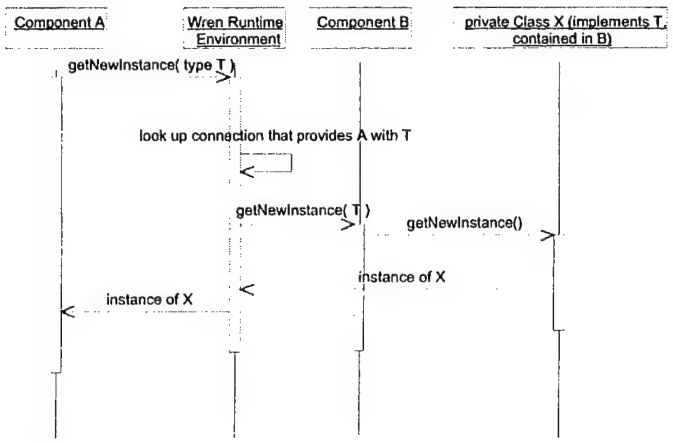


Figure 5. Instantiation across component boundaries in WREN.

Visual Café for instance-level configuration of the application.

WREN uses Remote Method Invocation (RMI) to communicate with a Visual Café plug-in, which automatically loads the components into the component library of Visual Café, from where they can be dragged into Visual Café's visual editor.

3.5 Execution

After the application has been configured, it can be executed. For testing, it can be executed in the WREN environment; this allows an iterative build process of alternating phases of configuring and executing. To make it possible to run the application outside of WREN, the environment can generate a configuration file that stores the names and URLs of the components that participate in this application, and how they are configured (their connections and the name of the main method). The WREN Runtime Environment, a jar file of 25 kilobytes, can then be used to run the application. It will first download the components if they are not yet locally available.

When an application is being executed and a component implementation needs to instantiate a data type that is specified by one of its requires ports (i.e. a data type that is implemented by another component), it will query the runtime environment for a new instance of that data type. Figure 5 depicts this process using a UML sequence diagram. The runtime environment checks the configuration of the application to determine which *provides* port is connected to this *requires* port. Then it asks the component that owns the *provides* port for a new instance, and returns it to the component that needs it. After instantiation, the component uses regular method calls to communicate with the other one; the polymorphism of the programming language makes it possible for one object to use another object without knowing the latter's precise type.

This mechanism constitutes an efficient runtime implementation of the concept of explicit connectors. The connection is explicit, because no component ever knows which of the other components provides the data type that it is using. At the same time it avoids most of the runtime overhead of message passing or similar decoupling strategies. An overhead occurs only when a data type is being instantiated, not each time its instances are used.

3.6 Component Evolution

When a component is marked as selected, the downloaded self-description can be implemented in one of two ways to provide access to the implementation of the component. In the usual case, it downloads a copy of the implementation and caches it locally. Then, it subscribes with the repository for update notifications. When an updated version of the component is published at the repository, the component is notified and can update itself.

The other possible strategy is service reuse [10]. Analogous to a client-server application architecture, the downloaded part of the component forwards requests to the master copy of the component that is located at the repository. Since the component is encapsulated, the difference between the two strategies is transparent to the user of the component, and thus to WREN. This means that the component can decide at runtime which strategy to use. For example, when the network transfer rate is high enough, the most current data can be directly accessed on the remote server. At times when the network is overloaded, the component can decide to use the locally cached version.

Both these strategies realize reuse by reference. In both cases, a logical connection between the application using a component and the original copy of the component is created in order to prevent the maintenance problems associated with reuse.

3.7 Summary

As described above, WREN implements the key requirements that we identified for a CBDE in Section 2:

- It supports modularity by accessing component implementations only through ports.
- It leverages component self-description by extending the design elements of Java Beans.
- It uses the global namespace of the Java language for its interfaces.
- It enables the process of application composition, including searching, selecting, and configuring of components. Component development is not supported, but acceptable support is provided by traditional development environments.
- It facilitates component connection with an easy-to-use graphical design editor. Adaptation is not yet supported.
- It provides multiple views to show the various stages of the application composition process.
- It realizes reuse by reference through integration with a component repository.

Thus, while WREN borrows from a number of isolated component concepts and techniques, we believe that WREN represents the first attempt to systematically support a diverse collection of concepts and techniques in an environment for lifecycle-wide component-based development.

4. RELATED WORK

While CBDEs have yet to become a focus of widespread research, there are several previous research efforts that contribute technologies, principles and insights for CBDE design.

An overview of the history and possible future of software engineering environments is given by Harrison et al. [11]. They consider *multi-view software environments* to be one of the most promising recent trends. Every complex system has many concerns that have to be considered separately. This is best done by providing different, independent views of the various aspects of a system. Type and instance view in WREN are examples of two views that show different aspects of the same system.

The ArchStudio project [17], which evolved out of the Arcadia project [13] and work on the C2 architectural style [33], defines an event-based architecture for a family of software engineering environments. The architectural style used lends itself to distribution, but it is still a subject of current research to determine whether this is possible on an Internet scale. However, integration of WREN with ArchStudio is planned. While tool integration in WREN is currently implemented on an ad-hoc basis, the principled approach of ArchStudio is clearly preferable.

Inscape [27] is an integrated development environment that uses formal module specifications to ensure component compatibility. In a similar way to our approach, it does not perform a complete semantic analysis, but instead leverages a well-designed namespace. However, rather than employing name equivalence over a namespace of interfaces, it employs partial semantic equivalence over a namespace of predicates used to specify interfaces. Inscape relies on Habermann and Perry's concept of well-formed compositions [8]. They list several desirable properties of component-based configurations, most of which are implicitly assured by WREN.

Koala [23] is a component model for embedded software in consumer electronics. It uses an explicit, visual description of architectures based on the architecture description language Darwin [15]. Like Darwin, it has *provides* and *requires* interfaces and treats interfaces as first-class entities. While Darwin was originally geared towards distributed systems, Koala demonstrates the usefulness of these features in a reuse-oriented component model.

The Application Web [22] is a strategy for sharing information between cooperating organizations that tries to minimize the problems caused by copying over organizational borders. To achieve this, connections are created to reuse data. Connections make it possible to automate caching, and to access all (not just part of) the context in which the data were originally created. Connections are comparable to the component references discussed in this paper.

The Basic Interoperability Data Model (BIDM) [4], developed by the Reuse Library Interoperability Group (RIG), is a standard for repositories of reusable artifacts that interoperate. The aim is to provide access to all artifacts offered by a network of repositories through any one repository, thus building a decentralized repository. There are two preconditions for this: There has to be a standard for meta-information about the artifacts, and a way to uniquely identify artifacts. The proposed data model covers some of the aspects we are suggesting for component self-description; however, the information is not stored in the component itself. Uniform Resource Names (URN) are the proposed solution for the identification problem; since a standard for URNs has not been adopted yet, URLs are used. In this way, the naming scheme is effectively equivalent to the naming conventions for Java packages that we rely on.

Whitehead et al. [35] point out that a well-designed architecture is an essential prerequisite for any component marketplace. They identify criteria for such an architecture, the most important of which are realized in WREN as follows:

- *Multiple component granularities* are given in WREN through the possibility to encapsulate any number of classes into a component.
- *Substitutability of components* is realized through the exclusive use of Java interfaces to specify component dependencies. Every interface can be implemented by any number of components, so that every component is substitutable.
- *Easy distribution of components* from seller to buyer is realized by the integration of development environment and component repository.

Brownsword et al. [5] share our view that new processes for developing component-based systems must be defined. Similar to Morisio et al. [21], they stress that the use of licensed components whose source code cannot be modified influences both requirements and design. Since there is a trade-off between the choice of components to license and the requirements and design of the system, these three issues have to be considered simultaneously.

Alpha Services [10] make applications available through the Internet. Instead of downloading and installing a program, services are accessed through the network when needed. This is a kind of reuse by reference; instead of components, services are reused. Candidates for Alpha Services are functionalities that are hard to develop, infrequently used, and can be modeled as transactions, such as natural language translation or large-scale optimization.

The Software Dock [9] is a system supporting the software deployment lifecycle. It integrates producer-side activities such as releasing and retiring a product with consumer-side activities such as installing, updating and uninstalling. Similar to WREN, a permanent connection is established between consumer and producer side. The Software Dock uses SRM [12] to administer the dependencies among application parts, which in WREN are administered by the components themselves. Similar to a CBDE, SRM is geared towards applications made up from independently produced parts.

5. CONCLUSIONS

In this paper we have motivated the need for a new generation of software environments to support the special needs of component-based development. We identified seven important requirements for CBDEs, and we described a prototype environment called WREN that we are building to implement these requirements and to provide a basis for further evaluation and study of the role of environment technology in component-based development.

There are several issues that remain to be resolved. Type-based adaptation does not exist yet in our prototype. Current tools provide mechanisms to adapt component instances, but not components themselves. We expect that the same methods of internal and external adaptation can be used in varied forms for type-based adaptation. Integration with development environments is another issue. It remains to be seen if tight integration of the CBDE with a

commercial development environment is the optimal solution, or if an alternative solution is needed.

Updating of components still requires manual effort. While the environment can automatically retrieve updates, it cannot update components that are being used in an application. Doing so will require support for dynamic architecture modification [24]. Another important issue is contract negotiation. A component may be able to dynamically decide about trade-offs between quality of service and price, for example, so that it can negotiate with another component or a human who wants to use this component. Negotiating will require explicit environment support, so that a user can define minimum requirements, policies, and so on.

6. ACKNOWLEDGMENTS

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; by the National Science Foundation under grant number CCR-9701973; and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U. S. Government.

7. REFERENCES

[1] Argo/UML. Accessed June 2001 at http://argouml.tigris.org/.

[2] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. Making Components Contract Aware. *Computer* 32, 7 (1999), 38-45.

[3] Bosch, J. Adapting Object-Oriented Components. In *Object-Oriented Technology.* Springer, Berlin, 1998, 379-383.

[4] Browne, S. V., and Moore, J. W. Reuse Library Interoperability and the World Wide Web. *Software Engineering Notes* 22, 3 (1997), 182-189.

[5] Brownsword, L., Oberndorf, T., and Sledge, C. A. Developing New Processes for COTS-Based Systems. *IEEE Software* 17, 4 (2000), 48-55.

[6] Dashofy, E. M., Medvidovic, N., and Taylor, R. N. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 3-12.

[7] DeRemer, F., and Kron, H. H. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2, 2 (1976), 80-86.

[8] Habermann, A. N., and Perry, D. E. System Composition and Version Control for Ada. In *Software Engineering Environments.* North-Holland, Amsterdam, 1981, 331-343.

[9] Hall, R. S., Heimbigner, D., and Wolf, A. L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 174-183.

[10] Harrison, C. G., and Stern, E. H. Alpha Services—An Experiment in Developing Enterprise Applications. Accessed June 2001 at http://www.isr.uci.edu/events/twist/twist2000/statements/harrison-stern.doc. 2000.

[11] Harrison, W., Ossher, H., and Tarr, P. Software Engineering Tools and Environments: A Roadmap. In *The Future of Software Engineering.* ACM, New York, 2000, 261-277.

[12] van der Hoek, A., Hall, R. S., Heimbigner, D., and Wolf, A. L. Software Release Management. In *Proc. Sixth European Software Engineering Conference.* Springer, Berlin, 1997, 159-175.

[13] Kadia, R. Issues Encountered in Building a Flexible Software Development Environment—Lessons from the Arcadia Project. *Software Engineering Notes* 17, 5 (1992), 169-180.

[14] Krueger, C. W. Software Reuse. *ACM Computing Surveys* 24, 2 (1992), 131-183.

[15] Magee, J., and Kramer, J. Dynamic Structure in Software Architectures. *Software Engineering Notes* 21, 6 (1996), 3-14.

[16] Maurer, P. M. Components: What If They Gave a Revolution and Nobody Came? *Computer* 33, 6 (2000), 28-34.

[17] Medvidovic, N., Oreizy, P., Taylor, R. N., Khare, R., and Guntersdorfer, M. An Architecture-Centered Approach to Software Environment Integration. Accessed June 2001 at ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf. 2000.

[18] Medvidovic, N., and Rosenblum, D. S. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Software Architecture.* Kluwer, Boston, 1999, 161-182.

[19] Medvidovic, N., Rosenblum, D. S., Robbins, J. E., and Redmiles, D. F. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology,* to appear.

[20] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[21] Morisio, M., Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E., and Condon, S. E. Investigating and Improving a COTS-Based Software Development Process. In *Proc. 2000 International Conference on Software Engineering.* ACM, New York, 2000, 32-41.

[22] Murer, T., and Van De Vanter, M. L. Replacing Copies with Connections: Managing Software across the Virtual Organization. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99).* IEEE Computer Society, Los Alamitos, 1999, 22-29.

[23] van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics So ftware. *Computer* 33, 3 (2000), 33-85.

[24] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

[25] Orso, A., Harrold, M. J., and Rosenblum, D. S. Component Metadata for Software Engineering Tasks. In *Proc. 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*. Springer, Berlin, 2000, 126-140.

[26] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (1972), 1053-1058.

[27] Perry, D. E. The Inscape Environment. In *Proc. 11th International Conference on Software Engineering*. IEEE, Washington, 1989, 2-12.

[28] Robbins, J. E., and Redmiles, D. F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology* 42, (2000), 79-89.

[29] Rosenblum, D. S., and Natarajan, R. Supporting Architectural Concerns in Component-Interoperability Standards. *IEE Proceedings-Software* 147, 6 (2000), 215-223.

[30] Selic, B., and Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems. Accessed June 2001 at http://www.rational.com/media/whitepapers/umlrt.pdf. 1998.

[31] Shaw, M., and Garlan, D. *Software Architecture*. Prentice Hall, Upper Saddle River, 1996.

[32] Szyperski, C. *Component Software*. ACM, New York, 1997.

[33] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22, 6 (1996), 390–406.

[34] Visual Café 4. Accessed June 2001 at http://www.webgain.com/products/visual_cafe/.

[35] Whitehead, E. J., Robbins, J. E., Medvidovic, N., and Taylor, R. N. Software Architecture: Foundation of a Software Component Marketplace. In *Proc. First International Workshop on Architectures for Software Systems*. ACM, New York, 1995, 276-282.

[36] Zaremski, A. M., and Wing, J. M. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology* 4, 2 (1995), 146-170.

from the International Workshop on Principles of Software Evolution—IWPSE 2001, Vienna, Austria, 2001

The Evolution of Software Evolvability

Chris Lüer David S. Rosenblum André van der Hoek

Institute for Software Research
University of California, Irvine
Irvine CA 92697-3425, USA
{chl,dsr,andre}@ics.uci.edu

ABSTRACT

We analyze two trends that have influenced the evolvability of component-based applications: increase of component exchangeability and increase of component distance. Exchangeability mechanisms can be classified either as code reuse or as service reuse. Component distance can vary from file scope to Internet scope. We discuss the various stages of evolvability in these dimensions, describe the state of the art, and speculate on future developments.

Keywords

Evolvability, exchangeability, code reuse, service reuse.

1. INTRODUCTION

Maintenance, or evolution, is the longest and most expensive phase of the software product lifecycle. Once released, software has to be corrected and updated. Evolvability is the property of programs that can easily be updated to fulfill new requirements; software that is evolvable will cost less to maintain. A component-based application is evolvable if it is easily possible to exchange individual components without changing others.

Component reuse exacerbates the problem of maintenance [10]. An application that consists of a large number of independently bought components will be much harder to update than a traditional, monolithic application, since each component will have individual updates from its manufacturer, and manufacturers will be independent from each other and located all over the world.

In this paper, we take a look at two historical trends that have made applications more evolvable. We identify several stages in each of these trends, including past stages that are often considered as outdated, and stages that many consider to be state of the art, but that are not widely used yet.

2. CLASSIFICATION

Two trends have influenced the development of evolvability technologies: an increase of component exchangeability and an increase of component distance (see Figure 1 for an overview). Component exchangeability means that components can easily be exchanged for other components, or updated with newer versions. Ideally, it should not be necessary to change other components to do this, or to change the architecture of the application. Component distance means the physical distribution of components over networks.

A consequence of these trends is the shift from design-time evolvability techniques to deployment-time evolvability techniques [8]. Design-time approaches to evolution require the source code to be accessible, but do not increase the evolvability of the compiled system. Deployment-time approaches, on the other hand, allow evolution to be managed by a component user without source access.

2.1 Component Exchangeability

The means to increase component exchangeability has been to introduce additional levels of indirection between components, i. e., to decouple them by making their connections more dynamic. The typical trade-off of this is a performance overhead. Additionally, application complexity increases, while component complexity decreases. Applications become harder to understand through the various indirection and dynamism techniques used; but individual components become more decoupled from each other, more independent and easier to identify, and thus they become cognitively less complex.

Increase of component exchangeability has happened in two contexts: code reuse and service reuse. We distinguish three stages of exchangeability in code reuse, and three stages of exchangeability in service reuse. While service reuse is a different approach, it provides a higher level of exchangeability than code reuse and can be regarded as an extension of code reuse techniques.

2.1.1 Code Reuse

Code reuse means that an application reuses a component by accessing its actual code (whether in source or in compiled form), loading it into memory, and then executing it. The application controls where, when and how the component is executed.

The first stage of component exchangeability in code reuse is no exchangeability at all. This is the consequence of reusing code by copy-and-paste—an arbitrary piece of source code from the old

Exchangeability (Code Reuse)	(1) Copy and Paste	(2) Static Linking	(3) Dynamic Linking		
Exchangeability (Service Reuse)			(1) RPC	(2) Messaging	(3) Broadcasting
Distance	File		File System	Internet	

Figure 1. Comparison of the evolution of component exchangeability and component distance. Time moves from left to right; corresponding stages are shown on top of each other.

project is copied and then pasted into the source of the new project. No connection is established between the original and the copy, and the copied code is not delimited in any way as to be recognizable as having been copied. There is nothing to keep the developer from changing the reused code once it has been pasted; this means that it may not be possible to identify the code as reused even if the source files are compared line by line later on. As a result, maintenance effort multiplies: each copy of the code will have to be maintained separately; the copies will evolve into separate directions and become more and more dissimilar over time. The advantage of this form reuse is that it requires only a minimum of tool support and does not increase program complexity or lower performance.

The second stage of exchangeability of components is exemplified by statically linked libraries, as they are usually used in the language C, for example. Libraries are distinct, well-defined units, or modules, but they are copied into each executable file. To exchange or update them, the application has to be relinked, which requires access to the compiler output files and the configuration of the application (as embodied in a make script, for example). Maintenance has become easier, but still requires rebuilding of the whole program whenever a library is modified. If a library is used by several applications, each of these has to be rebuilt when the library is updated. The advantage over copy-and-paste is that modules are clearly identified (at least on the source level), and that there is at most one copy of each reused code piece in each program. The trade-off is the need for a more complicated programming system.

Dynamic linking [3] constitutes the third stage of exchangeability; Java is an example. Each module is stored in its own file and exists only once per file system, and is accessed by all programs that need it. To update a module, one only has to exchange the corresponding file; it is not necessary to touch the actual application. Alternatively, it may be possible to update an environment variable (e. g., the Java class path) to point to the new version of a module instead of the old version. Besides added complexity, dynamic linking entails a performance trade-off: each module has to be linked to the application at run-time before it can be used.

Dynamic linking as used in Java avoids redundant copies of the same module in the scope of the file system. In a networked or distributed system, one may still have to cope with multiple copies at the various locations. It is possible, however, to extend dynamic linking to work on an Internet scale [6]. While the performance overhead becomes large, it can be reduced significantly through caching and event notification: the local system keeps a copy of the module, and is notified by the server that owns the original copy of the module whenever it is updated. In this way, the module has to be downloaded through the network only once after each update. The advantage of dynamic linking on an Internet scale is that only one master copy of each module needs to exist worldwide. Once the master copy is updated, the update is automatically promoted to all systems that use the module.

2.1.2 Service Reuse

The same historical evolution towards increasing exchangeability of components as with code reuse exists with service reuse mechanisms. Service reuse [5] means that the application is not granted access to the reused code, and thus cannot link to it, but instead it has to communicate with an independently running instance of it. We distinguish three stages in the development of service reuse technologies.

The first stage of exchangeability in service reuse is procedure call communication. Components are accessed when needed, for example through a remote procedure call. When the call is finished, the connection to the component is severed. For each subsequent call, a new connection has to be established. With dynamic linking, components can be updated independently from the applications that they are used in, but the update might not be effective unless the application has been restarted. Each component is loaded into memory after it is needed for the first time, and stays loaded until the application is shut down. With service access, whichever component is installed when the service call is made will be used. The performance overhead that is incurred is that of a remote procedure call. If a given service is used rarely, the overhead will be lower than with dynamic linking, but if it is used often, the overhead can be significantly higher.

Message based communication is the second stage of service reuse. Service reuse through procedure calls avoids linking the components while they are not communicating, but over the duration of the call, the components cannot be exchanged. Messaging makes components exchangeable at all times [9]. If an appropriate messaging infrastructure is provided, messages can be stored and resent in the case that the receiver is temporarily unavailable or does not respond. The performance overhead of message passing is significantly larger than of procedure calls. Also, it increases program complexity, because mechanisms to handle asynchronously arriving messages have to be present; programs cannot rely on messages arriving in a given order.

The third stage of exchangeability in service reuse is broadcasting, or implicit invocation [7]. Whereas messaging as in stage 2 is point-to-point communication with a limited number of receivers, broadcasting means that the application that requires a service sends this request as an event to all other applications. If one of them is able to fulfill the request, it returns a reply. The effect of

broadcasting is that the number and availability of service providers is completely transparent to the requesting application. A publish- and subscribe-mechanism can make this more efficient, but the overhead still includes all the overhead of message passing plus the overhead that is created through the potential multiplication of service providers; i. e., each service may be provided by more applications than necessary.

2.1.3 Discussion

Service reuse provides a higher level of exchangeability than code reuse, but its use is limited. It can provide data, or the results of computations, but it has only limited facilities to provide new data types or new behavior, as code reuse can. Service reuse is instance-oriented, whereas code reuse is type-oriented. As a consequence, service reuse is only practical for rarely used services that return results with a simple structure. It requires all data types to be converted to those data types that are known to the common platform of both communicating components (character strings are typically used); since no code is exchanged, custom data types cannot be used. Often-used services or services with complexly structured results will have to be integrated with the application as code. The tight coupling between components that service reuse avoids is traded off with a tight coupling between the components and their common platform. At the same time, a certain amount of functionality has to be replicated, since it has to be available at both communication partners, which makes maintenance harder. For these reason the required commonalties between the platforms should be kept as small as possible.

Java Remote Method Invocation [11] is an interesting combination of code reuse and service reuse. It is a language-specific remote procedure call mechanism and it can automatically load code that is not available at the destination of a call, but is necessary to execute the call. This typically happens when the call has parameters with polymorphic types. As with RPCs, the time during which the application is connected to the called component is limited to the duration of the call; as with dynamic linking, complex data types can be used for communication. The disadvantage of RMI compared to RPCs is that it does not support interoperability; both communication partners have to be Java programs. Its disadvantage versus dynamic linking is that it still requires objects to be marshalled; apart from the overhead, this means that object identity is lost.

2.2 Increase of Component Distance

Historically, the physical distance between components has increased. Increased distance usually causes looser coupling, because communication costs increase with distance. Thus, increase in component distance leads to an increase in application evolvability.

The trend of increasing component distance is linked to the increase in component exchangeability that was described above. Generally, the more distant two components are, the more exchangeable they are. This is caused by the fact that geographically distant systems are often administered by different people, making it necessary to be able to exchange or update components independently. We distinguish three stages of component distance: file, file system, and network.

In the first stage, all components of an application are contained inside one file. This corresponds to stages one and two of code reuse. Service reuse at file scope is simple procedure calls between modules in the same file; since this is not possible without code reuse of stage one or two, we do not consider this a service reuse stage of its own. To exchange a component, the file has to be rebuilt. The distance between components is zero.

In the second stage, components are spread out over a file system, which can either be local or distributed. Here, components can be exchanged by file system operations (such as moving, copying and deleting files), which are typically much more accessible and usable than the various functions of compilers, linkers, and similar tools that are needed to rebuild files in stage one.

The third stage of distance is the network, i. e., a system of multiple file systems that are owned and administered by different organizations. Components can be anywhere in a local or wide-area network. Reuse through networks that are not spanned by a file system is still rare or experimental. Applets and mobile agents are examples of code reuse here; various Internet protocols provide service reuse on a wide-area scale. In this stage, applications have to deal with high communication cost, potential network failures, and potential unavailability of components, so that coupling between components is typically low. Because of the global nature of the Internet, network communication protocols are typically highly standardized, so that individual components of a distributed application can easily be exchanged.

3. FUTURE DIRECTIONS

The current state of the art in code reuse is dynamic linking on a file-system scale. The discussion above shows a path to the future direction of evolution: dynamic linking on an Internet scale, as described above. This will move installation and maintenance effort from the local system administrator to the manufacturer of the component. The development of "self-installing" components fits well into another trend of software technology, the trend to put more and more information into components. Components contain not only code, but also assertions, documentation, and other forms of self-description. In the same way, components will be able to install themselves through the network. Complex applications composed out of independent components will be hard to maintain; dynamic linking on an Internet scale will automate most of the maintenance tasks.

WREN [6] is a prototypical component-based development environment developed by us that supports component self-description and Internet-wide dynamic linking. WREN allows an application developer to search for components in remote repositories, select components, compose them into an application, and execute the application. It maintains a logical connection to the master copy of a component in its repository, so that it can retrieve updates automatically.

Internet-wide dynamic linking as implemented in WREN is similar to the way Web pages are accessed. There is only one logical copy of each Web page. Pages are not copied to local systems; instead, clients always access the original page through the network. Pages can be cached to improve performance, but this happens internally and is completely transparent to all parties.

The current state of service reuse is procedure call or message passing; scaleable broadcast systems are still experimental [1] [2]. They are, however, a prerequisite of dynamic linking on an Internet scale, since the linker has to be notified of component updates.

For many systems, code reuse will not be possible, either because the amount of code and data is too large to be transferred or because code and data change too quickly. This is often the case with services that are provided through the Internet; for example, search engines, library catalogs, or weather forecasts. In these cases, service reuse is the only possibility. Since broadcasting is the form of service reuse that provides the highest degree of evolvability, we believe that systems will evolve in this direction.

It is obvious that the trend is going towards systems that are more and more distributed. The increasing availability of services on the Internet, wireless computing, and ubiquitous computing all work in this direction.

Strong mobile code [4] may turn out to be the fourth stage in code reuse. Strong mobile code is code that can change its own location in a network while it is executing, and the execution state is moved together with the code. While dynamic linking establishes a connection to the reused component for all of the execution time of the application, mobile code technologies might make it possible to reduce the connection time. It might be possible to combine strong mobility and broadcasting (the highest identified stage of service reuse) in the same way that RMI combines dynamic linking and remote procedure calls.

4. CONCLUSION

We believe that reusable software components are a promising technology. But to make reuse happen, composition mechanisms must provide for application evolvability. Off-the-shelf components that cannot be maintained will not be used.

The trends we described show the direction into which the construction of evolvable applications is evolving. Dynamic linking, event broadcasting, and Internet-wide distributed programs have been recognized as the next good things before, but only putting them into a historical framework shows the driving factors behind this development.

Further, this framework helps to identify areas of future research. It seems to be promising to work on identifying the next stages in this evolution.

5. ACKNOWLEDGMENTS

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; by the National Science Foundation under grants number CCR-9701973 and CCR-0093489; and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U. S. Government.

REFERENCES

[1] Carzaniga, A. *Architectures for an Event Notification Service Scalable to Wide-Area Networks.* PhD Thesis. Politecnico di Milano, Milan, 1998.

[2] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems* 19, 3 (2001), 332-383.

[3] Franz, M. Dynamic Linking of Software Components. *Computer* 30, 3 (1997), 74-81.

[4] Fuggetta, A., Picco, G. P., and Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering* 24, 5 (1998), 342-361.

[5] Harrison, C. G., and Stern, E. H. Alpha Services—An Experiment in Developing Enterprise Applications. Accessed January 2002 at http://www.isr.uci.edu/events/twist/twist2000/statements/harrison-stern.doc. 2000.

[6] Lüer, C., and Rosenblum, D. S. Wren—An Environment for Component-Based Development. *Software Engineering Notes* 26, 5 (2001), 207-217.

[7] Notkin, D., Garlan, D., Griswold, W. G., and Sullivan, K. Adding Implicit Invocation to Languages: Three Approaches. In *Object Technologies for Advanced Software.* Springer, Berlin, 1993, 489-510.

[8] Oreizy, P. Decentralized Software Evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1).* Kyoto, 1998.

[9] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

[10] Voas, J. Maintaining Component-Based Systems. *IEEE Software* 15, 4 (1998), 22-27.

[11] Waldo, J. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency* 6, 3 (1998), 5-7.

Supporting architectural concerns in component interoperability standards

D.S.Rosenblum and R.Natarajan

Abstract: There has been considerable work in industry on the development of component-interoperability models, such as COM, CORBA and JavaBeans. These models are intended to reduce the complexity of software development and to facilitate reuse of off-the-shelf components. The focus of these models is syntactic interface specification, component packaging, intercomponent communication, and bindings to a runtime environment. What these models lack is a consideration of architectural concerns—specifying systems of communicating components, explicitly representing loci of component interaction, and exploiting architectural styles that provide well understood global design solutions. The work described involves introducing support for architectural concerns in component models, particularly studying techniques to support notions of architectural style and explicit connectors. The JavaBeans component model has been enhanced to support component composition according to the C2 architectural style. The approach enables the design and development of applications in the C2 style using off-the-shelf Java components or 'beans' that are available to the designer. The techniques underlying the approach are described, along with a composition environment called 'ARABICA' that embodies these techniques. A number of important issues that must be addressed when extending component standards to support architectural concerns are identified.

1 Introduction

There has been considerable work in industry on the development of component interoperability models, such as COM [1], CORBA [2] and JavaBeans [3]. These models are intended to reduce the complexity of software development and to facilitate reuse of off-the-shelf (OTS) components [4]. Component interoperability models are also a necessary first step toward realisation of a viable software component marketplace. The focus of these models is syntactic interface specification, component packaging, intercomponent communication protocols, and mechanisms for binding to features of the runtime environment.

What these models lack is a consideration of architectural concerns – specifying systems of communicating components, explicitly representing loci of component interaction, and exploiting architectural styles that provide well understood global design solutions [5]. This is a deficiency that has also long plagued programming languages [6]. As research in software architecture has demonstrated, notions of architectural style and explicit connectors in the architecture of a system have the potential to provide important benefits in capturing system-level architectural properties that are carried through to implementation [5, 7]. Explicit connectors encapsulate the mechanisms by which components communicate and interoperate. An architectural style defines:

(*a*) a 'vocabulary' of design elements, namely component and connector types;
(*b*) 'design rules' that determine the permitted compositions of those elements;
(*c*) a 'semantic interpretation' giving compositions a well defined meaning; and
(*d*) 'analyses' that can be performed on systems built in the style [8].

The goal of the present work is to provide a way of incorporating support for architectural concerns in component interoperability models. Numerous benefits in providing such support can be identified. First, the properties and benefits of the architectural style chosen for a system can be preserved as detailed development of the system is undertaken with the component interoperability model. Further, during maintenance of the system, changes can be applied to the system in a way that preserves stylistic properties or informs developers whenever the changes violate the style; this would help prevent the occurrence of 'architectural drift' [5]. Explicit connectors allow greater decoupling of components, thereby offering the potential to increase the scalability of the component interoperability model to large compositions of components. Explicit connectors can also be used to support runtime architectural changes [7].

The JavaBeans component interoperability model was chosen as the initial platform for investigation. This choice was made for a variety of reasons:

(*a*) The Java language and the JavaBeans component model, as well as Sun's component model for distributed objects in business applications, Enterprise JavaBeans [9, 10], are all being widely adopted as *de facto* standards for

IEE Proceedings online no. 20000913

DOI: 10.1049/ip-sen:20000913

Paper first received 12th June and in revised form 23rd November 2000

The authors are with the Department of Information and Computer Science, University of California, Irvine, Irvine, CA 92697-3425, USA
E-mail: {dsr,rema}@ics.uci.edu
Rema Natajaran is currently employed at Cap Gemini Ernst & Young.

component-based software.
(*b*) The model of composition in JavaBeans is natural and straightforward.
(*c*) JavaBeans is a lightweight and flexible model that lends itself to modification, extension and experimentation.
(*d*) There are several JavaBeans tools and resources that are free or have negligible cost.

In addition, the C2 architectural style has been chosen as the initial architectural technology, because it is a novel style that is highly flexible and lends itself naturally to a variety of application domains and to specialised architectural operations, such as dynamic architectural change. C2 is one of many possible architectural styles that could have been chosen, yet it is nicely representative of the kinds of structural and behavioural rules and constraints that are imposed by the many architectural styles that have been formulated and studied.

In this paper, initial results and experiences in enhancing the JavaBeans component model to support component composition according to the C2 architectural style are reported. The approach permits the design and development of applications in the C2 style using off-the-shelf Java components or 'beans' that are available to the designer. The creation of individual components with their particular interfaces, functionalities and behaviours is a different task from architecture-based construction of a system to satisfy the system's requirements. The merging of the component interoperability model with the architectural style provides a seamless integration of both activities.

2 JavaBeans component model

JavaBeans is a component interoperability model tailored to the Java programming language. Interoperability is achieved primarily by designing component or 'bean' interfaces according to a 'component design pattern'. (The term 'design pattern' has been used to characterise the JavaBeans interface design convention, even though it does not correspond to the more prevalent notion of a design pattern as a frequently recurring program design solution [11].) The JavaBeans design pattern defines a naming scheme and interaction protocol to which compliant beans must adhere. The interface constituents governed by this design pattern include properties, methods and events that together define a bean interface. 'Properties' encapsulate key attributes of a bean. Properties are used by designers to customise a bean upon instantiation in a system, and by other beans to query and modify bean attributes at runtime. Properties can be 'simple', 'bound' (meaning that they generate events whenever their values change) or 'constrained' (meaning that their changes can be vetoed by other beans). 'Methods' are public operations that form part of the bean interface. Beans communicate with each other through bean 'events'; event handling is based on the Java JDK 1.1 event model.

The JavaBeans design pattern defines a notion of 'bean customisers' (which can be used to provide more complex customisation of a bean's behavior and visual appearance), and 'property editors' (which define more extensive editors that are used by designers to customise specific bean properties). These two mechanisms aid the design and implementation of generic beans that can easily be customised for different applications, thereby facilitating a measure of component reuse. Apart from supporting design and customisation of individual beans, the JavaBeans design pattern facilitates the use of visual design tools that dynamically 'introspect' beans and determine the capabilities they advertise through their interfaces, in order to permit incremental composition of beans. In particular, beans are visually composed in a 'beanbox environment' to create running applications, and their runtime behaviour can be customised and tested incrementally as the composition is created. This capability blurs the distinction between 'design time' and 'runtime', since manipulating beans in this manner actually has the effect of creating running instances of bean classes that co-operate according to the designer's intent.

The Sun Microsystems Beans Development Kit (BDK) includes a simple beanbox environment for developing beans using the JavaBeans design pattern and for instantiating and testing bean compositions [12]. This environment is representative of the kinds of visual design environments that can be used to support construction of applications with the JavaBeans component model.

The JavaBeans component model concentrates on specifying the syntactic interface that a Java software building block can or should present. It does not specify rules governing how the building blocks can or should be combined to create any kind of application. It specifies how two or more beans can communicate information, without imposing any semantic rules on the information exchanged or on the topology of any bean composition [3].

3 C2 architectural style

The C2 architectural style is a general system of architectural design rules suitable for a broad range of distributed applications in which (potentially heterogeneous) components interact in asynchronous fashion by exchanging events [13, 14]. Space limitations preclude a thorough presentation of C2, so the reader is referred to Taylor *et al.* for a complete description of the style [13]. Additional papers describe DRADEL and C2SADEL, a language and environment for specifying C2-style architectural models from which Java code templates can be generated [15, 16]; an approach to supporting architectural dynamism in C2-style architectures [7]; and an approach to reusing off-the-shelf middleware in C2-style architectures [17].

The building blocks of C2 architectures are 'components' (computational elements) and 'connectors' (flexible interconnection and communication elements). This separation of computation from communication permits the construction of flexible, extensible and scalable systems that can evolve both at design time and runtime. The style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing the use of multiple interoperability technologies in an architecture. This flexibility enables the event-based interoperability of JavaBeans to be used in C2-style architectures.

Central to the C2 style is the principle of 'substrate independence' – components are arranged in a layered fashion in a C2 architecture, and a component does not depend on components that reside beneath it in the vertical arrangement of component layers. Components communicate only by exchanging messages through connectors, which greatly simplifies the problem of control integration and facilitates low-cost interchangeability of components to construct different members of the same system family. Components cannot assume that they will execute in the same address space as other components. This eliminates complex dependencies, such as components sharing global variables, and it simplifies the modification of architec-

tures. Conceptually, components run in their own thread(s) of control, allowing components with different threading models to be integrated into a single application. (Note that while compositions of JavaBeans run in a single thread of control, individual beans do not exploit the availability of a shared address space. Hence the present approach to supporting the rules of the C2 style in JavaBeans is compatible with C2's conceptual model of each component executing in its own thread of control. Furthermore, distributed execution of beans in multiple threads and address spaces is supported by Enterprise JavaBeans.) Thus, a conceptual C2 architecture can be instantiated in a number of different ways, and many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation decisions.

C2 components and connectors have a notion of a 'top interface' and a 'bottom interface' through which they receive and send messages and communicate with other components in an architecture. This notion of top and bottom is important for ensuring substrate independence. The top interface represents the services the component requires from components above it in the architecture, and the bottom interface represents the services the component provides to components below it in the architecture. This model guides the architect in fitting OTS components into the context of the C2 architecture. It also aids further refinement and explicit description of the interface the component provides and the roles the component plays in the context of the architecture. Messages that travel up the architecture are called 'requests', and messages that travel down the architecture are called 'notifications'. Components execute application logic and communicate with other components in the architecture via requests and notifications. Components do not communicate directly with one another, but instead must communicate through connectors that take care of the management of message traffic in the system. Connectors, on the other hand, can be directly connected to each other.

Fig. 1 presents the wrapping model that has been devised for the C2 style to facilitate reuse of OTS components as 'black boxes' [18]. An OTS component (the Internal Object in Fig. 1) is wrapped so that all interaction

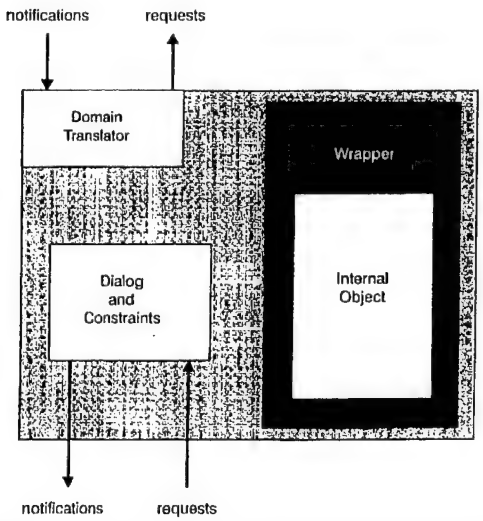


Fig. 1 *Wrapping of OTS components in C2-style architectures. General model of wrapping for C2*

between the OTS component and the rest of the architecture is through requests and notifications handled by the Dialog element, while the Domain Translator and Constraints elements are concerned primarily with alleviating architectural mismatches [19]. The Domain Translator is used to resolve incompatibilities between communicating components such as mismatches between message names, parameter types and ordering of parameters. The Constraints element specifies constraints that cannot be violated by the component, provides recovery mechanisms when constraints are violated and exceptions are raised, and provides mechanisms to customise the component so that constraints can be satisfied without raising exception conditions. This model is exploited in the present approach to composing JavaBeans according to the rules of the C2 style.

4 ARABICA: A C2-aware composition environment

The investigation of the problem of supporting architectural concerns in component models was begun by enhancing the BDK beanbox described in Section 2. In the present approach, beans are created or reused according to the JavaBeans design pattern, thus retaining all the advantages of the beans component model. However, the JavaBeans model is extended to incorporate the notion of components and connectors as defined in the C2 architectural style, and the beanbox composition functionality is extended to enforce the rules of the C2 style. Thus, the beanbox has been enhanced to make it 'C2-aware'. The resulting environment is called 'ARABICA'.

4.1 Instantiating beans in the C2 style

ARABICA provides two mechanisms to instantiate beans as C2 components and connectors—defining subclasses of classes provided in the C2-class framework, and wrapping OTS beans using the C2 wrapping mechanism.

For the first mechanism, the C2 class framework [13], a Java implementation framework for C2-style systems, is adapted for use as Java beans. These classes can be extended to create beans that are intrinsically C2-ready. When these beans are instantiated in the beanbox, they automatically publish their C2 interface and can thus be hooked up 'as is' to compose applications. In the current implementation of ARABICA, the connectors of a C2-style architecture constructed in ARABICA must be instances of (subclasses of) the framework class C2Connector.

When implementing a C2 component bean with the C2 class framework in this fashion, the component designer may be limiting the reusability of the bean for use in other applications with different architectural styles. The designer may thus instead prefer to use the complete power of the beans component model independently of the C2 class framework, allowing greater reuse of the bean in other applications and architectures.

Hence, for the second mechanism, the C2 wrapping mechanism has been adapted to create C2 components from OTS beans. Fig. 2 presents the model of component wrapping used, which follows the general model of wrapping presented in Fig. 1. OTS beans that have been independently developed by third-party vendors can be instantiated into ARABICA. On instantiation, ARABICA creates a wrapper object for the bean to make it C2 compliant. This C2 wrapper is created with the help of an interactive dialogue that presents the interface of the bean to the designer and guides the designer in mapping

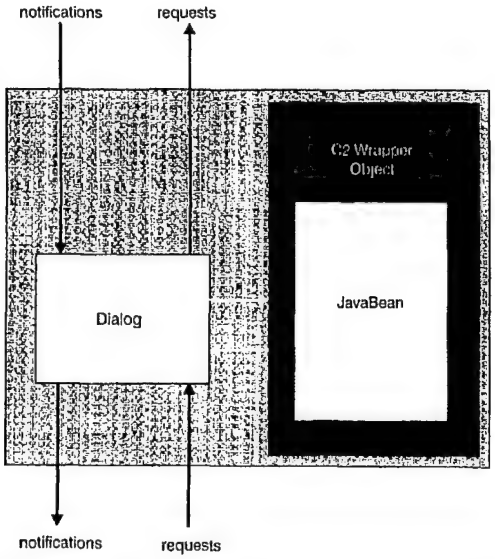


Fig. 2 *Wrapping of OTS components in C2-style architectures. Wrapping model as used in ARABICA*

the bean's events into C2 requests and notifications. A particular bean event can be tagged as both a request and as a notification in the architecture. ARABICA then uses this information to build the internal Dialog element that is responsible for converting incoming requests and notifications into bean events. On receipt of a request (from the bottom port) or a notification (from the top port) the Dialog element extracts the bean event. If the bean is interested in receiving that event, then the appropriate method in the bean is invoked with the event. In addition, the Dialog element translates all of the bean's generated events into requests and notifications and sends them to the appropriate ports (top or bottom).

Most of the translation required for converting beans into C2 components involves mapping bean events to requests and notifications in the C2 style. This is carried out under the assumption that bean communication occurs via events. The properties that a bean publishes in its property sheet are used 'as is' after the bean has been wrapped as a C2 component. Other embellishments provided for manipulation of beans such as property editors and bean customisers can also be used as is in ARABICA.

4.2 Composing beans in ARABICA

C2-compliant beans can be created using either of the two approaches described above and then instantiated into ARABICA. ARABICA incorporates all the C2 style rules and constraints into its composition capabilities. It provides a C2 Style Dialog that notifies designers whenever stylistic constraints are violated during design and thus guides the designer through the composition process. Components and connectors are hooked up according to the intended architecture of the system using the JavaBeans visual-wiring mechanism, where request events and notification events become the two kinds of events that beans use to interoperate. As required by the C2 style, components cannot be wired directly to other components, and hence connector beans must be instantiated to handle the propagation of events. Thus, unlike in the traditional beans model, components composed in ARABICA do not maintain internal lists of the other components with which they interact. Instead, event notification is handled by the C2 connector beans. Hence, components are more strongly decoupled, and their behaviour is better confined to the execution of application logic. ARABICA thus allows one to build complex compositions of beans in the C2 style.

4.3 ARABICA implementation details

ARABICA is implemented as an extension of the beanbox of Sun Microsystems' BDK version 1.0. The beanbox is entirely implemented in Java and comprises roughly 9000 lines of Java code. Supporting enforcement of the C2 style in ARABICA required the addition of roughly 4000 lines of Java. Roughly half the new code implements the checking of the C2 style rules on a composition, and the other half implements the generation of wrappers according to Fig. 2. It is not yet possible to formulate any generic principles or systematic approach for introducing style-based enhancements to a design environment as has been done in ARABICA. This experience is mirrored in other work on architectural design environments. For instance, Aesop is a system for generating style-dependent architectural modelling environments that allows an environment builder to define a style in terms of component, connector and connection rule types [20]. However, Aesop does not provide an architectural language for defining styles and generating environments. Instead, the environment builder must work at the programming language level (specifically Tcl/Tk) to customise the environment according to an intuitive understanding of the rules defined by the style. Similarly, Argo is a family of design environments that uses a system of 'critics' to provide feedback to the designer about design anomalies [21]. Instances of Argo have been built for C2 and for UML, but these were custom implementations crafted for their respective design domains [21, 22].

5 Example JavaBeans-based C2 architecture

The approach is illustrated with a telephone network simulator application; construction of the application in ARABICA is depicted in Fig. 3. (The vertical attachments between components and connectors are presented in Fig.3 for the sake of clarity. In actuality, the wiring paths between beans are visible neither in ARABICA nor in JavaBeans visual environments in general.) The system consists of telephones, local switches and longdistance switches. Each of these components is provided as a bean that publishes events (such as ring, hang-up, busy) and properties (such as phone number and area code). The properties are bound properties and thus fire Property Change events, permitting use of property changes as the primary mechanisms for driving the execution of the simulator. In the C2 architecture for the telephone system, the telephones form the lowest layer of the architecture (i.e. the 'interface elements', as is typical of C2-style architectures), with local switches in a layer above the telephones, and the long distance switches at the highest layer.

As described in Section 4, on instantiation of the beans into ARABICA, each bean is wrapped in a C2 wrapper. The Dialog element of the C2 wrapper dynamically introspects the bean and then displays the bean's list of events to guide the designer in mapping the events to requests and notifications. For example, for the telephone component the designer would select the 'dial' event as a request that needs to travel 'up' the architecture, and for the local-

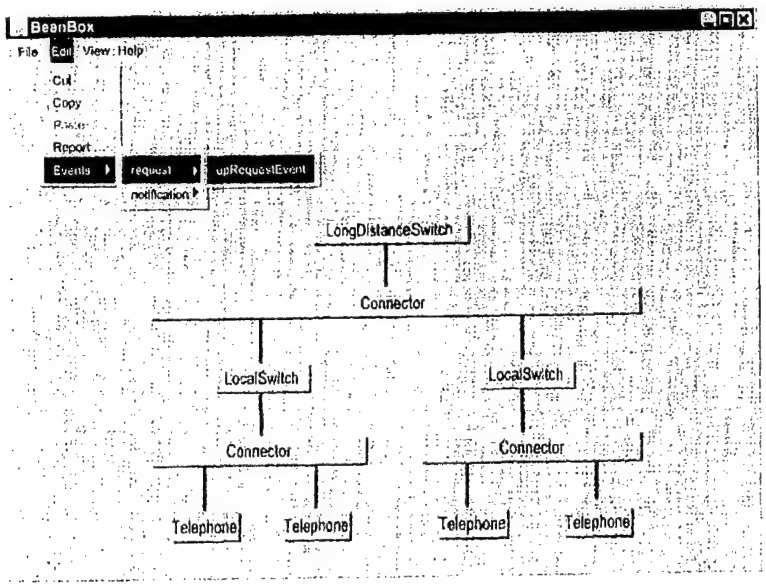


Fig. 3 *Composition of a C2-style telephone network application in ARABICA*

switch component the designer would select the 'ring' event as a notification that needs to travel 'down' the architecture.

The standard connectors provided by the C2 class framework are used to link the components of the telephone simulator together. Each connector propagates request events fired by a component connected to its bottom interface to all components attached to its top interface. Thus, a request by a telephone to dial a number is propagated through the connector above it to the local switch that handles requests for the calling telephone's area code (specified as a property of the local switch). The local switch in turn forwards the request to the long distance switch above it if the call is to a different area code. The long distance switch then sends the message down the architecture as notifications to the local switches below it. The local switch with the area code for the dialled number processes the notification by generating a 'ring' event as a notification for the telephones below it. The telephones receive the notification, and the telephone with

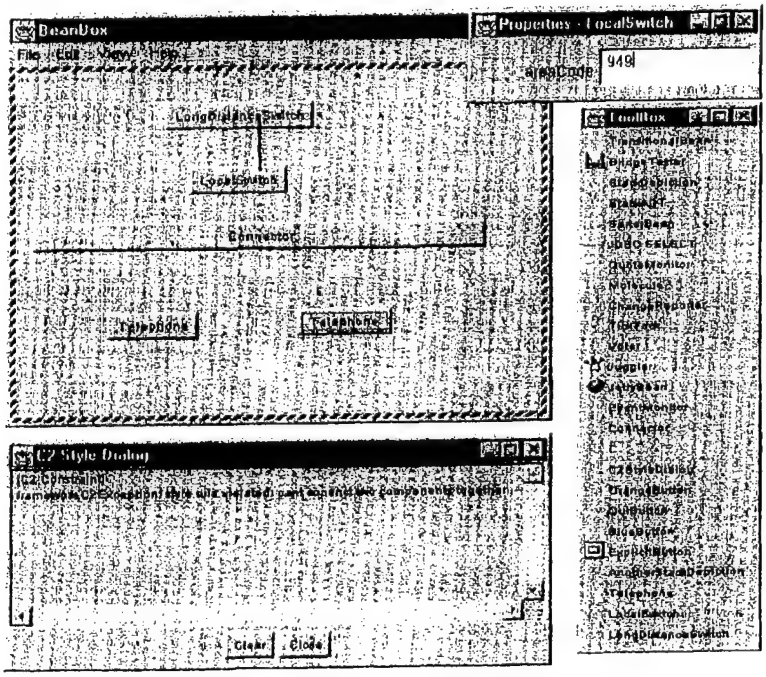


Fig. 4 *Wiring up the long distance switch and the local switch in ARABICA, which throws a C2-style violation exception in a C2-style dialog*

the correct number (specified as a property of the telephone) responds to the notification by processing it. The beans themselves retain their original component interfaces and implementations without modification, but the wrapper ensures that the beans can communicate effectively with the rest of the architecture.

As shown in Fig. 4, ARABICA operates in the same manner as other beanbox environments: beans are instantiated into the architecture using a drag-and-drop visual style of composition, and the properties of the currently selected bean are displayed in a property sheet. However, as the telephone network is built by plugging beans into the architecture, ARABICA also checks to ensure that the C2 style rules are honoured. For example, an attempt to link two components directly raises an exception message in a popup window, notifying the designer of a rule violation. Fig. 4 shows how an attempt to link two components (the long distance switch and the local switch) generates a violation that brings up the C2-style dialog.

6 Discussion

This paper presents some of the earliest known results in which principles of architecture-based development have been incorporated into standard component technologies. While initial efforts with the JavaBeans component model may appear at first glance to have been fairly straightforward, the experience described in Sections 4 and 5 has permitted identification of a number of interesting technical issues in bringing architectural concerns to component technologies. Some of these issues are addressed in full or in part in the current version of ARABICA, while others will be the subjects of future study.

The JavaBeans component model was originally designed for composing small GUI objects into tightly knit groups of communicating beans to form a GUI for a system. (Interestingly, the C2 architectural style was also initially formulated as an architectural style for GUIs [13]. Subsequent experience with C2 has demonstrated its broader applicability.) Using the model entails building event management directly into every bean, thereby limiting their reusability, increasing their dependence on other beans, and limiting the scalability of compositions. Because the present approach is based on the C2 style, it is possible to address this problem by using explicit connectors to enable a clean separation between application logic and communication logic. The connectors encapsulate logic for event broadcasting, event filtering, and other interaction, making it easier to support substitution of components and connectors, and dynamic alteration of the architecture. Thus, it has been possible to strengthen the JavaBeans model with respect to the decoupling of components and the scalability of compositions.

To reap the potential economic advantages of reusing OTS components in a software system, the components typically need to be relatively large-grained and to have minimal dependencies on other components. Several interesting issues came up in the effort to create 'plug-and-play' functionality with large-grained OTS beans in ARABICA. For instance, if an OTS bean does not contain all of the functionality needed for a particular conceptual component in an architecture, alternative ways of providing this functionality must be devised [18]. One simple approach is to alter the source code of the OTS bean, but this breaks the plug-and-play paradigm and also requires access to the source code. Instead, other beans might be used in conjunction with the OTS bean (without altering the latter's source code) to provide the necessary functionality. In this case, it may make sense for the wrapper generated for the OTS bean to encapsulate the full set of beans rather than just the individual OTS bean. This set of beans would then implement the conceptual component in the context of the C2 architecture.

In general, this style of encapsulation essentially requires support for hierarchical refinement of a composition across multiple levels. While the JavaBeans model does not explicitly support a notion of hierarchical decomposition, it should be fairly easy to adapt ARABICA to provide such support. Such an adaptation should provide a seamless transition from the 'architecture world' (where stylistic issues may be paramount at higher levels of abstraction) to the 'components world' (where lower-level portions of the composition of components may be better constrained by the rules of design patterns [11]. Support for hierarchical decomposition in ARABICA could be usefully complemented by two composition modes— the ordinary, unconstrained mode of bean composition, and a mode constrained by the rules of a chosen architectural style.

As described in Section 3, the C2 model of wrapping OTS components recognises the need to handle various kinds of mismatches in the assumptions components make about their environment. If the interface expected by one OTS component does not match the interfaces provided by other components, a domain translator must be built for the first component to provide a mapping between the vocabularies of the expected and provided interfaces. The wrappers generated by ARABICA currently provide no support for domain translation. Additionally, if a composition of beans is being constructed according to a formal architectural model, it would be fruitful to provide mechanisms in a development and testing environment such as ARABICA to validate the instantiated architecture against the associated model. As described in Section 3, such mechanisms would naturally be embodied in the Constraints element of a C2 wrapper, but currently ARABICA's C2 wrapper provides no support for this.

There are numerous possibilities to be explored in strengthening support for design at the architectural level, apart from the work already carried out for the C2 style. Note that ARABICA currently requires OTS beans to be wrapped individually each time they are instantiated in an architecture. This is true even when the same bean is instantiated several times in an architecture, leading to possible inconsistencies in the way the instances are wrapped. A better option would be to create an 'architecture-template bean' from the wrapper the first time the wrapper is generated. Then, rather than generating wrappers for each subsequent instance of the OTS bean, the template bean would instead be instantiated and populated with an instance of the OTS bean, thereby guaranteeing uniformity among all wrapped instances of the OTS bean.

It is relevant to ask whether it makes sense to use just any arbitrary OTS bean in a C2 architecture. While it is possible syntactically to wrap any bean into a C2 architecture, semantically the bean may not be well suited to the C2 architectural style, especially if its implementation assumes the existence of other beans and makes direct access to the public data members and methods of those other beans 'outside' the confines of the interoperation paths established in the normal JavaBeans manner. There is nothing to prevent third-party components from providing or accessing public data members and methods. In particular, any direct access of a component's public data members and methods by another component is a violation

of C2's prohibition against direct component–component communication. And a situation in which the former component conceptually resides in the same or a lower layer of the architecture than the latter component would be a violation of C2's principle of substrate independence. While it may be possible to detect such situations and even provide architectural mechanisms to overcome them, in general it will not make sense to do this for all OTS beans. Even for OTS beans where it does make sense, such situations are not currently handled by ARABICA. One possible solution that is at present being explored is the use of 'stubs' to act as proxies for such accesses. Stubs for the accessed beans are generated and maintained by the wrapper of the OTS bean making the accesses, and the OTS bean makes calls to the stub to access a public data member or a method of the actual bean represented by the stub. For each call, the Dialog element of the wrapper creates a request or notification that captures the semantics of the call and sends it up or down the architecture as appropriate. The wrapper for the bean receiving this request or notification passes it to its own Dialog element. This Dialog in turn calls the appropriate method of the bean and then sends the result mapped appropriately to a request or notification. The bean that made the initial request then gets the result request or notification ultimately as a return value from the stub.

Less serious is the issue of mapping the events generated by a bean to C2 requests and notifications. When reusing an OTS bean as a C2 component, it might be useful for a particular event the bean generates to travel both up *and* down the architecture. This capability is supported by ARABICA, although it might seem to be a violation of the C2 rules at first glance. For example, consider a layered C2 architecture supporting two different GUI views of a data model that is stored and constructed from a persistent database. One of the view components may send a request to the data-model component to make a change to the database component. The data-model component would then send a notification down to all the view components to update their views with the change, and at the same time send a request to the database component to commit the change to the data. An OTS bean that is used as the data-model component may use just one event to represent a change to its internal model. Thus, in the C2 context, a request *and* a notification need to be generated for the same event.

Another issue worth studying is the extent to which we can adapt the visual approach to architectural construction for use on distributed architectures. A current limitation of the JavaBeans model is that it does not support visual composition of distributed beans that must communicate via remote procedure call (e.g. using Java Remote Method Invocation) or more sophisticated middleware such as CORBA or TIBCO's TIB/Rendezvous. With ARABICA, such mechanisms for distributed interaction within C2 connectors can naturally be incorporated, yet it is necessary to provide additional support for specifying a composition of beans that will execute across distributed hardware. Techniques are currently being explored for supporting OTS connectors in ARABICA, which will make possible the construction of distributed bean-based applications. These techniques will exploit the capabilities of Enterprise JavaBeans (EJB) as well as work on employing OTS middleware in C2-style architectures [17]. Note also that future versions of the Java™ 2 Platform Enterprise Edition (J2EE) will define an architecture for portable connectors, which will allow 'container' entities such as EJBs to integrate more easily with existing enterprise systems [10]. The possibility of using these J2EE connectors to implement the connectors of C2-style architectures will be investigated. An even better approach might be to support lightweight automated construction or customisation of context-dependent connector implementations.

A final issue that bears further study is the extent to which it is possible to generalise the work reported here to other architectural styles and component models. Of the various architectural styles that have been described in the literature (see Shaw and Garlan for a useful discussion of common architectural styles [23]), C2 imposes possibly the greatest number of rules and constraints, since it combines features of a number of more general styles (including the client/server, layered system and implicit invocation styles). In other words C2 can be viewed as a specialisation of these other styles, and so it is to be expected that it will be a simple matter to find ways of enforcing the rules of these less restrictive styles within a composition of components. As for generalising to other component models, not all component models lend themselves to a visual style of composition, which has been relied on fairly heavily in the present work with JavaBeans. However, given an initial look at COM and CORBA, it appears that it is possible to create appropriate mechanisms for incorporating support for architectural concerns in other models.

7 Related work

Section 4.3 discussed some related work on supporting architectural concerns in design environments, but there has been little work to date on supporting architectural concerns in conjunction with standard component technologies. Sullivan *et al.* used an architecture-based approach to understand Microsoft's COM component model [24]. In particular, rather than attempting to apply the rules of any particular style to COM, they instead viewed COM itself as an architectural standard and modelled its definition in the Z calculus to validate their understanding of COM, which allowed them to discover anomalies in the definition itself. Jazayeri *et al.* followed a similar approach to exploring the relationship between architecture and component technologies [25, 26]. In particular, they looked at an existing component framework, the C++ Standard Template Library, and identified the architectural style induced by that framework. In a similar vein, Di Nitto and Rosenblum have studied the architectural styles induced by middleware infrastructures [27].

C2, UniCon and Darwin are examples of ADLs that provide a proprietary implementation infrastructure to support an associated ADL. C2 provides its class framework as its infrastructure, and this class framework is implemented in multiple programming languages [13]. UniCon supports implementation generation for a predefined collection of connectors [28]. Darwin is supported by an infrastructure called Regis for distributed programs that are configured using Darwin [29, 30]. There has also been work in the Darwin project on supporting architectural modelling of CORBA-based systems [31]. In addition to providing ADL-specific infrastructure support, there has been work on incorporating substantial support for architectural modelling into the Unified Modelling Language (UML) a standard object-oriented-design notation [32].

There is also work by Stuurman on relating the tenets of software architecture to the features of JavaBeans, but her analysis is presented independently of any particular architectural style or ADL [33]. In contrast to the direction of

the present work, she concluded that the best use of architectural models with JavaBeans is to reverse engineer a model from an existing composition of beans.

8 Conclusions

Having considered and explored the possibility of combining a popular component interoperability model with a useful and representative software architectural style, the authors are encouraged that this approach can bring advantages to the development of component-based software, and believe that it will be fruitful to explore similar approaches with other architectural and component technologies. The philosophy of substrate independence in C2 makes composition of components fairly easy. It is possible to leverage the strengths of the JavaBeans component model and the ability of ARABICA to dynamically introspect the interface of a bean and map it to C2-style interactions. The use of a wrapper separates the application logic in the bean component from its communication with other architectural components.

A key advantage of the approach described in this paper is that a broad range of architectural design activities is now integrated into a single environment, from the design, implementation and adaptation of individual components to the design, implementation and integration of architectures that are compositions of these individual elements. This integration portends an architecture-based development approach that facilitates easy shifting of focus from one activity to another. Sophisticated architectural development tools built along these lines will tie in neatly with component-based software development.

In the future, we plan to investigate further the issues and opportunities opened up by this approach. In addition to the issues discussed in Section 6, the problem of architecture-based testing of component compositions is also of interest. The ability to test the runtime behaviour of bean components in a design environment would be extremely useful for testing different architectural configurations. A natural place to provide instrumentation support for testing and analysis is in the Dialog and Constraints elements of the C2 wrapper shown in Fig. 1. In a separate tool called 'ROBUSTA', work is also being performed to support checking of component semantic constraints. An excellent home for this work is the ADL C2SADEL and its associated environment DRADEL, which support modelling and evolution of C2 architectures according to a theory of heterogeneous typing of architectural elements [15, 16].

In summary, experience so far with JavaBeans, C2 and ARABICA is helping with the development and expansion of an understanding of the synergy between component models and software architectures.

9 Acknowledgments

Discussions with Elisabetta Di Nitto, Jacky Estublier, Alfonso Fuggetta, Neno Medvidovic, Peyman Oreizy, Kevin Sullivan and Dick Taylor helped to improve many of the ideas presented in this paper. We also thank the anonymous reviewers for their helpful criticisms.

This work was sponsored by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant F49620-98-1-0061; and by the National Science Foundation under grant CCR-9701973. The US Government is authorised to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the US Government.

10 References

1 BOX, D.: 'Essential COM' (Addison Wesley, Reading, MA, 1998)
2 SIEGEL, J.: 'CORBA fundamentals and programming' (Wiley, New York, 1996)
3 VANHELSUWE, L.: 'Mastering JavaBeans' (SYBEX Inc, 1997)
4 SZYPERSKI, C.: 'Component software: beyond object-oriented programming' (Addison Wesley, Reading, MA, 1998)
5 PERRY, D.E., and WOLF, A.L.: 'Foundations for the study of software architecture', *ACM SIGSOFT Softw. Eng. Notes*, 1992, **17**, (4), pp. 40–52
6 DEREMER, F., and KRON, H.H.: 'Programming-in-the-large versus programming-in-the-small', *IEEE Trans. Softw. Eng.*, 1976, **SE-2**, (2), pp. 80–86
7 OREIZY, P., MEDVIDOVIC, N., and TAYLOR, R.N.: 'Architecture-based runtime software evolution'. 20th international conference on *Software engineering*, April 1998, Kyoto, Japan
8 ABOWD, G., ALLEN, R., and GARLAN, D.: 'Using style to understand descriptions of software architecture'. ACM SIGSOFT '93 symposium on the *Foundations of software engineering*, December 1993, Redondo Beach, CA, pp. 9–20
9 FLANAGAN, D., FARLEY, J., CRAWFORD, W., and MAGNUSSON, K.: 'Java enterprise in a nutshell' (O'Reilly, 1999)
10 SHANNON, B., HAPNER, M., MATENA, V., DAVIDSON, J., PELEGRILLOPART, E., CABLE, L., and ENTERPRISE TEAM: 'Java™ 2 platform, enterprise edition: platform and component specifications' (Addison Wesley, Boston, 2000)
11 GAMMA, E., HELM, R., JOHNSON, R., and VLISSIDES, J.: 'Design patterns: Elements of reusable object-oriented software' (Addison-Wesley, Reading, 1995)
12 DESOTO, A. 'Using the Beans development kit 1.0: A tutorial'. JavaSoft, Sun Microsystems, Inc., 1997
13 TAYLOR, R.N., MEDVIDOVIC, N., ANDERSON, K.M., WHITEHEAD, E.J., ROBBINS, J.E., NIES, K.A., OREIZY, P., and DUBROW, D.L.: 'A component- and message-based architectural style for GUI software', *IEEE Trans. Softw. Eng.*, 1996, **22**, (6), pp. 390–406
14 OREIZY, P., MEDVIDOVIC, N., TAYLOR, R.N., and ROSENBLUM, D.S.: 'Software architecture and component technologies: bridging the gap'. Digest of the OMG-DARPA-MCC workshop on *Compositional software architectures*, 1998
15 MEDVIDOVIC, N., ROSENBLUM, D.S., and TAYLOR, R.N.: 'A type theory for software architectures'. Technical report UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, 1998
16 MEDVIDOVIC, N., ROSENBLUM, D.S., and TAYLOR, R.N.: 'A language and environment for architecture-based software development and evolution'. 21st international conference on *Software engineering*, May 1999, Los Angeles, pp. 44–53
17 DASHOFY, E.M., MEDVIDOVIC, N., and TAYLOR, R.N.: 'Using off-the-shelf middleware to implement connectors in distributed software architectures'. 21st international conference on *Software engineering*, May 1999, Los Angeles, pp. 3–12
18 MEDVIDOVIC, N., OREIZY, P., and TAYLOR, R.N.: 'Reuse of off-the-shelf components in C2-style architectures'. 19th international conference on *Software engineering*, May 1997, Boston, pp. 692–700
19 GARLAN, D., ALLEN, R., and OCKERBLOOM, J.: 'Architectural mismatch: why reuse is so hard', *IEEE Softw.*, 1995, **12**, (6), pp. 17–26
20 GARLAN, D., ALLEN, R., and OCKERBLOOM, J.: 'Exploiting style in architectural design environments'. ACM SIGSOFT '94 second symposium on the *Foundations of software engineering*, Dec. 1994, New Orleans, pp. 175–188
21 ROBBINS, J.E., HILBERT, D.M., and REDMILES, D.F.: 'Extending design environments to software architecture design', *Autom. Softw. Eng.*, 1998, **5**, (3), pp. 261–290
22 ROBBINS, J.E., and REDMILES, D.F.: 'Cognitive support, UML adherence, and XMI interchange in Argo/UML', *Inf. Softw. Technol.*, 2000, **42**, (2), pp. 79–89
23 SHAW, M., and GARLAN, D.: 'Software architecture: perspectives on an emerging discipline' (Prentice-Hall, 1996)
24 SULLIVAN, K.J., SOCHA, J., and MARCHUKOV, M.: 'Using formal methods to reason about architectural standards'. 19th international conference on *Software engineering*, May 1997, Boston, pp. 503–513
25 JAZAYERI, M.: 'Component programming – a fresh look at software components', Proceedings of the 5th European Software engineering conference, (Springer Verlag, 1995), pp. 457–478
26 GALL, H., JAZAYERI, M., KLÖSCH, R., and TRAUSMUTH, G.: 'The architectural style of component programming'. 21st annual international *Computer software and applications conference COMPSAC'97*,

Aug. 1997, Washington, DC, pp. 18–25
27 DI NITTO, E., and ROSENBLUM, D.S.: 'Exploiting ADLs to specify architectural styles induced by middleware infrastructures'. 21st international conference on *Software engineering*, May 1999, Los Angeles, pp. 13–22
28 SHAW, M., DELINE, R., KLEIN, D., ROSS, T., YOUNG, D., and ZELESNIK, G.: 'Abstractions for software architecture and tools to support them', *IEEE Trans. Softw. Eng.*, 1995, **21**, (4), pp. 314–335
29 MAGEE, J., DULAY, N., and KRAMER, J.: 'Regis: A constructive development environment for distributed programs', *Distrib. Syst. Eng.*, 1994, **1**, (5), pp. 304–312
30 MAGEE, J., and KRAMER, J.: 'Dynamic structure in software architectures'. ACM SIGSOFT '96 fourth symposium on the *Foundations of software engineering*, Oct. 1996, San Francisco, pp. 3–14
31 MAGEE, J., TSENG, A., and KRAMER, J.: 'Composing distributed objects in CORBA'. Third international symposium on *Autonomous decentralized systems*, April 1997, Berlin, pp. 257–263
32 ROBBINS, J.E., MEDVIDOVIC, N., REDMILES, D.F., and ROSENBLUM, D.S.: 'Integrating architecture description languages with a standard design method'. 20th international conference on *Software engineering*, April 1998, Kyoto, pp. 209–218
33 STUURMAN, S.: 'Software architecture and Java Beans'. First working IFIP conference on *Software architecture*, February 1999, San Antonio, pp. 183–199

from the Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), Davis, CA, November 2-3, 2000

Component Metadata for Software Engineering Tasks

Alessandro Orso[1], Mary Jean Harrold[1], and David Rosenblum[2]

[1] College of Computing
Georgia Institute of Technology
{orso,harrold}@cc.gatech.edu
[2] Information and Computer Science
University of California, Irvine
dsr@ics.uci.edu

Abstract. This paper presents a framework that lets a component developer provide a component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks throughout the software engineering lifecycle. The framework is defined in a general way, so that the metadata can be easily extended if new types of data have to be provided. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. We motivate the untapped potential of component metadata by showing the need for metadata in the context of testing and analysis of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application: generating self-checking code and program slicing.

Keywords: Components, component-based systems, distributed components, metadata.

1 Introduction

In recent years, component-based software technologies have been increasingly considered as necessary for creating, testing, and maintaining the vastly more complex software of the future. Components have the potential to lower the development effort, speed up the development process, leverage other developers' efforts, and decrease maintenance costs. Unfortunately, despite their compelling potential, software components have yet to show their full promise as a software engineering solution, and are in fact making some problems more difficult. The presence of externally-developed components within a system introduces new challenges for software-engineering activities. Researchers have reported many

problems with the use of software components, including difficulty in locating the code responsible for given program behaviors [4], hidden dependences among components [4,5], hidden interfaces that raise security concerns[12, 17], reduced testability [18], and difficulties in program understanding [4]. The use of components in a distributed environment makes all the above problems even more difficult, due to the nature of distributed systems. In fact, distributed systems (1) generally use a middleware, which complicates the interactions among components, and (2) involve components that have a higher inherent complexity (e.g., components in e-commerce applications that embody complex business logic and are not just simple GUI buttons).

Several of the above problems are due to the lack of information about components that are not internally developed. Consider an application developer who wishes to use a particular component by incorporating it into her application, either by using it remotely over a network or by interacting with it through middleware such as CORBA [6]. The application developer typically has only primary interface information supporting the invocation of component functions. In particular, she has no source code, no reliability or safety information, no information related to validation, no information about dependences that could help her evaluate impacts of the change, and possibly not even full disclosure of interfaces and aspects of component behavior. When the task to be performed is the integration of the component, information about the component interface and its customizable properties can be all that is needed. Other software engineering tasks, however, require additional information to be performed on a component-based system.

In this paper, we present a framework that lets the component developer provide the component user with different kinds of information, depending on the specific context and needs. The framework is based on presenting this information in the form of metadata. *Metadata* describe static and dynamic aspects of the component, can be accessed by the user, and can be used for different tasks. The idea of providing additional data together with a component is not new: It is a common feature of many existing component models, albeit a feature that provides relatively limited functionality. In fact, the solutions provided so far by existing component models are tailored to a specific kind of information and lack generality. To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

The framework that we propose is defined in a general way, so that the metadata can be easily extended to support new types of data. In our approach, we define a unique format and a unique tag for each kind of metadata provided. The tag lets the user of the component both treat the information provided as metadata in the correct way and query for a specific piece of information. Because the size and complexity of common component-based software applications are constantly growing, there is an actual need for automated tools to develop, integrate, analyze, and test such applications. Several aspects of the framework that we propose can be easily automated through tools. Due to the way the

metadata are defined, tools can be implemented that support both the developer who has to associate some metadata with his component and the user who wants to retrieve the metadata for a component she is integrating into her system.

We show the need for metadata in the context of analysis and testing of distributed component-based systems, and introduce our framework with the help of an example. We sketch a possible scenario consisting of an application developer who wants to perform two different software engineering tasks on her application. The first task is in the context of self-checking code. In this case, the metadata needed to accomplish the task consist of pre-conditions and post-conditions for the different functions provided by the component, and invariants for the component itself. This information is used to implement a checking mechanism for calls to the component. The second task is related to slicing. In this case, the metadata that the developer needs to perform the analysis consist of summary information for the component's functions. Summary information is used to improve the precision of the slices involving one or more calls to the component, which would otherwise be computed making worst-case assumptions about the behavior of the functions invoked.

The rest of the paper is organized as follows. Section 2 provides some background on components and component-based applications. Section 3 presents the motivating example. Section 4 introduces the metadata framework and shows two possible uses of metadata. Section 5 illustrates a possible implementation of the framework for metadata. Finally, Section 6 draws some conclusions and sketches future research directions.

2 Background

This section provides a definition of the terms "component" and "component-based system," introduces the main technologies supporting component-based programming, and illustrates the different roles played by developers and users of components.

2.1 Components

Although there is broad agreement on the meaning of the terms "component" and "component-based" systems, different authors have used different interpretations of these terms. Therefore, we still lack a unique and precise definition of a component. Brown and Wallnau [2], define a component as "a replaceable software unit with a set of contractually-specified interfaces and explicit context dependences only." Lewis [10] defines a component-based system as "a software system composed primarily of components: modules that encapsulate both data and functionality and are configurable through parameters at run-time." Szyperski [16] says, in a more general way, that "components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."

In this paper, we view a *component* as a system or a subsystem developed by one organization and deployed by one or more other organizations, possibly in different application domains. A component is open (i.e., it can be either extended or combined with other components) and closed (i.e., it can be considered and treated as a stand-alone entity) at the same time. According to this definition, several examples of components can be provided: a class or a set of cooperating classes with a clearly-defined interface; a library of functions in any procedural language; and an application providing an API such that its features can be accessed by external applications. In our view, a *component-based* system consists of three parts: the user application, the components, and the infrastructure (often called middleware) that provides communication channels between the user application and the components. The user application communicates with components through their interfaces. The communication infrastructure maps the interfaces of the user application to the interfaces of the components.

2.2 Component Technologies

Researchers have been investigating the use of components and component-based systems for a number of years. McIlroy first introduced the idea of components as a solution to the software crisis in 1968 [13]. Although the idea of components has been around for some time, only in the last few years has component technology become mature enough to be effectively used. Today, several component models, component frameworks, middleware, design tools, and composition tools are available, which allow for successful exploitation of the component technology, and support true component-based development to build real-world applications.

The most widespread standards available today for component models are the CORBA Component Model [6], COM+ and ActiveX [3], and Enterprise JavaBeans [7]. The CORBA Component Model, developed by the Object Management Group, is a server-side standard that lets developers build applications out of components written in different languages, running on different platforms, and in a distributed environment. COM+, OLE, and ActiveX, developed by Microsoft, provide a binary standard that can be used to define distributed components in terms of the interface they provide. The Enterprise JavaBeans technology, created by Sun Microsystems, is a server-side component architecture that enables rapid development of versatile, reusable, and portable applications, whose business logic is implemented by JavaBeans components [1]. Although the example used in this paper is written in Java and uses JavaBeans components, the approach that we propose is not constrained by any specific component model and can be applied to any of the above three standards.

2.3 Separation of Concerns

The issues that arise in the context of component-based systems can be viewed from two perspectives: the component developer perspective and the component

user (application developer)[1] perspective. These two actors have different knowledge, understanding, and visibility of the component. The component developer knows about the implementation details, and sees the component as a white box. The component user, who integrates one or more components to build a complete application, is typically unaware of the component internals and treats it as a black box. Consequently, developers and users of a component have different needs and expectations, and are concerned with different problems.

The component developer implements a component that could be used in several, possibly unpredictable, contexts. Therefore, he has to provide enough information to make the component usable as widely as possible. In particular, the following information could be either needed or required by a generic user of a component:

Information to evaluate the component: for example, information on static and dynamic metrics computed on the components, such as cyclomatic complexity and coverage level achieved during testing.

Information to deploy the component: for example, additional information on the interface of the component, such as pre-conditions, post-conditions, and invariants.

Information to test and debug the component: for example, a finite state machine representation of the component, regression test suites together with coverage data, and information about dependences between inputs and outputs.

Information to analyze the component: for example, summary data-flow information, control-flow graph representations of part of the component, and control-dependence information.

Information on how to customize or extend the component: for example, a list of the properties of the component, a set of constraints on their values, and the methods to be used to modify them.

Most of the above information could be computed if the component source code were available. Unfortunately, this is seldom the case when a component is provided by a third party. Typically, the component developer does not want to disclose too many details about his component. The source code is an example of a kind of information that the component developer does not want to provide. Other possible examples are the number of defects found in the previous releases of the component or the algorithmic details of the component functionality. Metadata lets the component developer provide only the information he wants to provide, so that the component user can accomplish the task(s) that she wants to perform without having knowledges about the component that are supposed to be private.

To exploit the presence of metadata, the component user needs a way of knowing what kind of additional information is packaged with a given component and a means of querying for a specific piece of information. The type of

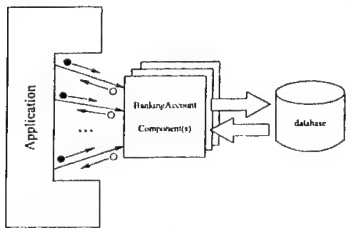
[1] Throughout the remainder of the paper, we use "component user" and "application developer" interchangeably.

information required may vary depending on the specific needs of the component user. She may need to verify that a given component satisfies reliability or safety requirements for the application, to know the impact of the substitution of a component with a newer version of the same component, or to trace a given execution for security purposes. The need for different information in different contexts calls for a generic way of providing and retrieving such information.

Whereas it is obvious that a component user may require the above information, it is less obvious why a component developer would wish to put effort into computing and providing it. From the component developer's point of view, however, the ability to provide this kind of information may make the difference in determining whether the component is or can be selected by a component user who is developing an application, and thus, whether the component is viable as a product. Moreover, in some cases, provision of answers may even be required by standards organizations — for instance, where safety critical software is concerned. In such cases, the motivation for the component developer may be as compelling as the motivation for the component user.

3 Motivating Example

In this section, we introduce the example that will be used in the rest of the paper to motivate the need for metadata and to show a possible use of this kind of information.



(a)

```
public class BankingAccount {
   ...
   BankingAccount(String code) {...}
   ...
   public void open()
             throws CantOpenException,
             InvalidPINException {...}
   public float getBalance() {...}
   public float withdraw(float amount) {...}
   public float deposit(float amount) {...}
   public float moveFunds(BankingAccount destacct,
                          float amount) {...}
};
```

(b)

Fig. 1. (a) High-level view of the application. (b) Interface of the `BankingAccount` component.

The example consists of part of a distributed application for remote banking. The application uses one or more instances of an externally-developed component to access a remote database containing the account-related information. Figure 1(a) provides a high-level view of the application, to show the interaction between the user code and the component(s). Figure 1(b) shows the subset of the `BankingAccount` component interface used by the application. We assume the common situation in which the component user is provided with the interface of the component together with some kind of user documentation.

```
...
public boolean checkingToSavings(String cAccountCode,
                                 String sAccountCode,
                                 float amount) {
1.  BankingAccount checking(cAccountCode);
2.  BankingAccount saving(sAccountCode);
3.  float balance, total;
    ...
4.  checking.open();
5.  saving.open();
    ...
6.  balance = checking.moveFunds(saving, amount);
    ...
7.  total = balance + additionalFunds;
    ...
}
...
```

Fig. 2. Fragment of the application code.

Figure 2 shows a fragment of the application code. The code is part of a method whose semantics is to move a given amount of money from a checking account to a savings account. The first two parameters of the method are two strings containing the codes of the checking account and of the savings account, respectively. The third parameter is a number representing the amount of the funds to be moved. Note that, for the sake of the presentation, we have simplified the example to make it smaller, self contained, and more understandable.

4 Metadata

When integrating an externally-developed component into a system, we may need to perform a set of tasks including, among possible others, the gathering of third-party certification information about the component, analyses and testing of the system, and assessment of some quality of the resulting application. These tasks require more than the mere binary code together with some high level description of the component's features. Unfortunately, the source code for the component is generally unavailable, and so is a formal specification of the component. Moreover, we are not simply interested in having a specific kind of information about the component, as a specification would be, but rather we need a way of providing different kinds of information depending on the context. This is the idea behind the concept of metadata: to define an infrastructure that lets the component developer (respectively, user) add to (respectively, retrieve from) the component the different types of data that are needed in a given context or for a given task. Obviously, metadata can also be produced for internally-developed components, so that all the components that are used to build an application can be handled in an homogeneous way.

This notion of providing metadata with software components is highly related to what electrical engineers do with hardware components: just as a resistor is not useful without its essential characteristic such as resistance value, tolerance, and packaging, so a software component needs to provide some information about itself to be usable in different context. The more metadata that are available from or about a component, the fewer will be the restrictions on tasks that can be performed by the component user, such as applicable program analysis techniques, model checking, or simulation. In this sense, the availability of metadata for a component can be perceived as a "quality mark" by an application developer who is selecting the components to deploy in her system.

Metadata range from finite-state-machine models of the component, to QoS[2]-related information, to plain documentation. In fact, any software engineering artifact can be a metadatum for a given component, as long as (1) the component developer is involved in its production, (2) it is packaged with the component in a standard way, and (3) it is processable by automated development tools and environments (including possibly visual presentation to human users).

As stated in the Introduction, the idea of providing additional data — in the form of either metadata or metamethods returning the metadata — together with a component is not new. The properties associated with a JavaBean [1] component are a form of metadata used to customize the component within an application. The *BeanInfo* object associated with a JavaBean component encapsulates additional kinds of metadata about the component, including the component name, a textual description of its functionality, textual descriptions of its properties, and so on. Analogously, the interface *IUnknown* for a DCOM [3] component permits obtaining information (i.e., metadata) about the component interfaces. Additional examples of metadata and metamethods can be found in other component models and in the literature [14, 20, 8]. Although these solutions to the problem of how to provide additional data about a component are good for the specific issues they address, they lack generality. Metadata are typically used, in existing component models, only to provide generic usage information about a component (e.g., the name of its class, the names of its methods, the types of its methods' parameters) or appearance information about GUI components (e.g., its background and foreground colors, its size, its font if it's a text component). To date, no one has explored metadata as a general mechanism for aiding software engineering tasks, such as analysis and testing, in the presence of components.

To show a possible situation where metadata are needed, let us assume that the component user that we met in Section 3 had to perform two different software engineering tasks on her application: implementation of a run-time checking mechanism and program slicing. We refer to the system in Figures 1 and 2 to illustrate the two tasks.

[2] Quality of Service

4.1 Self-checking Code

Suppose that the component user is concerned with the robustness of the application she is building. One way to make the system robust is to implement a run-time checking mechanism for the application [9, 15]. A run-time check mechanism is responsible for (1) checking the inputs of each call prior to the actual invocation of the corresponding method, (2) checking the outputs of each call after the execution of the corresponding method, and (3) suitably reacting in case of problems.

It is worth noting that these checks are needed even in the presence of an assertion-based mechanism in the externally-developed component. For example, the violation of an assertion could imply the termination of the program, which is a situation that we want to avoid if we are concerned with the robustness of our application. Moreover, according to the design-by-contract paradigm, a client should be responsible for satisfying the method pre-condition prior to the invocation of such method.

```
public class BankingAccount {
  //@ invariant ( ((balance > 0) || (status == OVERDRAWN)) && \
  //@             ((timeout < LIMIT) || (logged == false)) );

  public void open() throws CantOpenException,
                            InvalidPINException {
    //@ pre (true);
    //@ post (logged == true)
  }

  public float getBalance() {
    //@ pre (logged == true);
    //@ post ( ((return == balance ) && (balance >= 0)) || \
    //@         (return == -1.0) );
  }

  public float withdraw(float amount) {
    //@ pre ( (logged == true) && \
    //@       (amount < balance) );
    //@ post ( (return == balance' ) && \\
    //@         (balance' == balance - amount) );
  }

  public float deposit(float amount) {
    //@ pre (logged == true);
    //@ post ( (return == balance' ) && \\
    //@         (balance' == balance + amount) );
  }

  public float moveFunds(BankingAccount destination, float amount) {
    //@ pre ( (logged == true) && \
    //@       ((amount < 1000.0) || (userType == ADMINISTRATOR)) && \
    //@       (amount < balance) );
    //@ post ( (return == balance' ) && \
    //@         (balance' == balance - amount) );
  }
};
```

Fig. 3. Fragment of the component code.

The run-time checks on the inputs and outputs are performed by means of checking code embedded in the application. This code can be automatically generated by a tool starting from a set of pre-conditions, post-conditions, and invariants compliant with a given syntax understood by the tool. As an alternative, the checking code can be written by the application developer starting from the same conditions and invariants. A precise description of the way conditions and invariants can be either automatically used by a tool or manually used by a programmer is beyond the scope of this paper. Also, we do not discuss the possible ways conditions and invariants can be available, either directly provided by the programmer or automatically derived from some specification. The interested reader can refer to References [9] and [15] for details.

The point here is that, if the application developer wants to implement such a mechanism, she needs pre- and post-conditions for each method that has to be checked, together with invariants. This is generally not a problem for the internally-developed code, but is a major issue in the presence of externally-developed components. The checking code for the calls to the external component cannot be produced unless that external component provides the information that is needed. Referring to the example in Figure 1, what we need is for the `BankingAccount` component to provide metadata consisting of an invariant for the component, together with pre- and post-conditions for each interface method.

Figure 3 provides, as an example, a possible set of metadata for the component `BankingAccount`.[3] The availability of these data to the the component user would let her implement the run-time checks described above also for the calls to the externally-developed component. The task would thus be accomplished without any need for either the source code of the component or any other additional information about it.

4.2 Program Slicing

Program slicing is an analysis technique with many applications to software engineering, such as debugging, program understanding, and testing. Given a program P, a program *slice* for P with respect to a variable v and a program point p is the set of statements of P that might affect the value of v at p. The pair $< p, v >$ is known as a *slicing criterion*. A slice with respect to $< p, v >$ is usually evaluated by analyzing the program, starting from v at p, and computing a transitive closure of the data and control dependences.

To compute the transitive closure of the data and control dependences, we use a slicing algorithm that performs a backward traversal of the program along control-flow paths from the slicing criterion [11]. The algorithm first adds the statement in the slicing criterion to the slice and adds the variable in the slicing criterion to the, initially empty, set of *relevant variables*. As the algorithm visits a statement s in the traversal, it adds s to the slice if s may modify (define) the value of one of relevant variables v. The algorithm also adds those variables that

[3] For sake of brevity, when the value of a variable V does not change, we do not show the condition "V' = V" and simply use V as the final value instead.

are used to compute the value of v at s to the set of relevant variables. If the algorithm can determine that s definitely changes v, it can remove v from the relevant variables because no other statement that defines v can affect the value of v at this point in the program. The algorithm continues this traversal until the set of relevant variables is empty.

Referring to Figure 2, suppose that the application developer wants to compute a slice for her application with respect to the slicing criterion $<$ `total`, 7 $>$.[4] By inspecting statement 7, we can see that both `balance` and `additionalFunds` affect the value of `total` at statement 7. Thus, our traversal searches for statements that may modify `balance` or `additionalFunds` along paths containing no intervening definition of those variables. Because statement 6 defines `balance`, we add statement 6 to the slice. We have no information about whether `checking` uses its state or its parameters to compute the return value of `balance`. Thus, we must assume, for safety, that `checking`, `saving`, and `amount` can affect the return value, and include them in the set of relevant variables. Because `balance` is definitely modified at statement 6, we can remove it from the set of relevant variables. At this point, the slice contains statements 6 and 7, and the relevant variables set contains `amount`, `checking`, and `saving`.

When the traversal processes statement 5, it adds it to the slice but it cannot remove `saving` from the set of relevant variables because it cannot determine whether `saving` is definitely modified. Likewise, when the traversal reaches statement 4, it adds it to the slice but does not remove `checking`. Because the set of relevant variables contains both `checking` and `saving`, statements 1 and 2 are added to the slice and `cAccountCode` and `sAccountCode` are added to the set of relevant variables. When the traversal reaches the entry to `checkingToSavings`, traversal must continue along calls to this method searching for definitions of all parameters. The resulting slice contains all statements in method `checkingToSavings`.

There are several sources of imprecision in the slicing results that could be improved if some metadata had been available with the component. When the traversal reached statement 6 — the first call to the component — it had to assume that the state of `checking` and the parameters to `checking` were used in the computation of the return value, `balance`. However, an inspection of the code for `checking.moveFunds` shows that `saving` does not contribute to the computation of `balance`. Suppose that we had metadata, provided by the component developer, that summarized the dependences among the inputs and outputs of the method.[5] We could then use this information to refine the slicing to remove some of the spurious statements.

Consider again the computation of the slice for slicing criterion $<$ `total`, 7 $>$, but with metadata for the component. When the traversal reaches statement 6, it

[4] Also assume that the omitted part of the code are irrelevant to the computation of the slide.

[5] We may be able to get this type of information from the interface specifications. However, this kind of information is rarely provided with a component's specifications.

uses the metadata to determine that `saving` does not affect the value of `balance`, and thus does not add `saving` to the set of relevant variables at that point. Because `saving` is not in the set of relevant variables when the traversal reaches statement 5, statement 5 is not added to the slice. Likewise, when the traversal reaches statement 2, statement 2 is not added to the slice. Moreover, because `saving` is not added to the slice, `sAccountCode` is not added to the set of relevant variables. When the traversal is complete, the slice contains only statement 1, 3, 4, 6, and 7 instead of all statements in method `checkingToSavings`. More importantly, when the traversal continues into callers of the method, it will not consider definitions of `sAccountCode`, which could result in many additional statements being omitted from the slice. The result is a more precise slice that could significantly improve the utility of the slice for the application developer's task.

5 Implementation of the metadata framework

In this section, we show a possible implementation of the metadata framework. The proposed implementation provides a generic way of adding information to, and retrieving information from, a component, and is not related to any specific component model. To implement our framework we need to address two separate issues: (1) what format to use for the metadata, and (2) how to attach metadata to the component, so that the component user can query for the kind of metadata available and retrieve them in a convenient way.

5.1 Format of the Metadata

Choosing a specific format suitable for all the possible kind of metadata is difficult. As we stated above, we do not want to constrain metadata in any way. We want to be able to present every possible kind of data — ranging from a textual specification of a functionality to a binary compressed file containing a dependence graph for the component or some kind of type information — in the form of metadata. Therefore, for each kind of metadata, we want to (1) be able to use the most suitable format, and (2) be consistent, so that the user (or the tool) using a specific kind of metadata knows how to handle it.

This is very similar to what happens in the Internet with electronic mail attachment or file downloaded through a browser. This is why we have decided to rely on the same idea behind MIME (Multi-purpose Internet Mail Extensions) types. We define a metadata type as a tag composed of two parts: a type and a subtype, separated by a slash. Just like the MIME type "application/zip" tells, say, a browser the type of the file downloaded in an unambiguous way, so the metadata type "analysis/data-dependence" could tell a component user (or a tool) the kind of metadata retrieved (and how to handle them). The actual information within the metadata can then be represented in any specific way, as long as we are consistent (i.e., as long as there is a one-to-one relation between the format of the information and the type of the metadatum).

By following this scheme, we can define an open set of types that allows for adding new types and for uniquely identifying the kind of the available data. A metadatum is thus composed of a header, which contains the tag identifying its type and subtype, and of a body containing the actual information. We are currently investigating the use of XML [19] to represent the actual information within a metadatum. By associating a unique DTD (Document Type Definition) to each metadata type, we would be able to provide information about the format of the metadatum body in a standard and general way. We are also investigating a minimum set of types that can be used to perform traditional software engineering tasks, such as testing, analysis, computation of static and dynamic metrics, and debugging.

5.2 Producing and Consuming Metadata

As for the choice of the metadata format, here also we want to provide a generic solution that does not constrain the kinds of metadata that we can handle. In particular, we want to be as flexible as possible with respect to the way a component developer can add metadata to his component and a component user can retrieve this information. This can be accomplished by providing each component with two additional methods: one to query about the kinds of metadata available, and the other to retrieve a specific kind of metadata. The component developer would thus be in charge of implementing (manually or through a tool) these two additional methods in a suitable way. When the component user wants to perform some task involving one or more externally-developed components, she can then determine what kind of additional data she needs, query the components, and retrieve the appropriate metadata if they are available.

Flexibility can benefit from the fact that metadata do not have to be provided in a specific way, but can be generated on-demand, stored locally, stored remotely, depending on their characteristics (e.g., on their amount, on the complexity involved in their evaluation, on possible dependences from the context that prevent summarizing them). As an example, consider the case of a dynamically-downloaded component, provided together with a huge amount of metadata. In such a situation, it is advisable not to distribute the component and the metadata at the same time. The metadata could be either be stored remotely, for the component to retrieve them when requested to, or be evaluated on demand. With the proposed solution, the component developer can choose the way of providing metadata that is most suitable for the kind of metadata that he is adding to the component. The only constraint is the signature of the methods invoked to query metadata information and to retrieve a specific metadatum, which can be easily standardized.

5.3 Metadata for the Example

Referring to the example of Section 3, here we provide some examples of how the above implementation could be developed in the case of an application built using JavaBeans components.

We assume that the `BankingAccount` component contains a set of metadata, among which are pre-conditions, post-conditions, and invariants, and data-dependence information. We also assume that the methods to query the component about the available metadata and to retrieve a given metadatum follows the following syntax:

```
String[] component-name.getMetadataTags()
Metadata component-name.getMetadata(String tag, String[] params)
```

When the application developer acquires the component, she queries the component about the kind of metadata it can provide by invoking the method `BankingAccount.getMetadataTags()`. Because this query is just an invocation of a method that returns a list of the tags of the available metadata, this part of the process can be easily automated and performed by a tool (e.g., an extension of the JavaBeans BeanBox). If the tags of the metadata needed for the tasks to be performed (e.g., `analysis/data-dependency` and `selfcheck/contract`) are in the list, then the component user can retrieve them. She can retrieve the invariant for the component by executing

`BankingAccount.getMetadata("selfcheck/contract", params)`,

where `params` is an array of strings containing only the string "invariant," and obtain the post-condition for `getBalance` by executing

`BankingAccount.getMetadata("selfcheck/contract", params)`,

where `params` is an array of strings containing the two strings "post" and "getBalance." Similar examples could be provided for the retrieving of the other information to be used for the tasks.

Our intention here is not to provide all the details of a possible implementation of the framework for a given component model, but rather to give an idea of how the framework could be implemented in different environments, and how most of its use can be automated through suitable tools.

6 Conclusion

In this paper, we have motivated the need for various kinds of metadata about a component that can be exploited by application developers when they use the component in their applications. These metadata can provide information to assist with many software engineering tasks in the context of component-based systems. We focused on testing and analysis of components, and with the help of an example discussed the use of metadata for two tasks that a component user might want to perform on her application: generating self-checking code and program slicing. In the first case, the availability of metadata enabled the task to be performed, whereas in the second case, it improved the accuracy and therefore the usefulness of the task being performed. These are just two examples of the kinds of applications of metadata that we envision for distributed component-based systems.

We have presented a framework that is defined in a general way, so to allow for handling different kinds of metadata in different application domains. The framework is based on (1) the specification of a systematic way of producing and

consuming metadata, and (2) the precise definition of format and contents of the different kinds of metadata. This approach will ease the automated generation and use of metadata through tools and enable the use of metadata in different contexts.

Our future work includes the identification and definition of a standard set of metadata for the most common software engineering activities, and an actual implementation of the framework for the JavaBean component model.

Acknowledgments

Gregg Rothermel provided suggestions that helped the writing of the paper. The anonymous reviewers and the workshop discussion supplied helpful comments that improved the presentation. This work was supported in part by NSF under NYI Award CCR-0096321 and ESS Award CCR-9707792 to Ohio State University, by funds from the State of Georgia to Georgia Tech through the Yamacraw Mission, by a grant from Boeing Aerospace Corporation, by the ESPRIT Project TWO (Test & Warning Office - EP n.28940), by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061, and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

1. Javabeans documentation. http://java.sun.com/beans/docs/index.html, October 2000.
2. A. W. Brown and K. C. Wallnau. Enginnering of component-based systems. In A. W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1996.
3. N. Brown and C. Kindel. *Distributed Component Object Model protocol: DCOM/1.0.* January 1998.
4. R. Cherinka, C. M. Overstreet, and J. Ricci. Maintaining a COTS integrated solution — Are traditional st atic anallysis techniques sufficient for this new programming methodology? In *Proceedings of the International Conference on Software Maintenance*, pages 160–169, November 1998.
5. J. Cook and J. Dage. Highly reliable ungrading components. In *Proceedings of the 21st International Conference on Software Engine ering*, pages 203–212, May 1999.
6. The common object request broker: Architecture and specification, October 2000.

7. Enterprise javabeans technology. http://java.sun.com/products/ejb/index.html, October 2000.
8. G. C. Hunt. Automatic distributed partitioning of component-based applications. Technical Report TR695, University of Rochester, Computer Science Department, Aug. 1998. Tue, 29 Sep 98 18:13:17 GMT.
9. N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, 1990.
10. T. Lewis. The next $10,000_2$ years, part II. *IEEE Computer*, pages 78–86, May 1996.
11. D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 421–430. IEEE Computer Society Press, 1999.
12. U. Lindquist and E. Jonsson. A map of security risks associated with using cots. *IEEE Computer*, 31(6):pages 60–66, June 1998.
13. D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.
14. G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–14. The USENIX Association, 1999.
15. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
16. C. Szyperski. *Component Oriented Programming.* Addison-Wesley, first edition, 1997.
17. J. Voas. Maintaining component-based systems. *IEEE Software*, 15(4):22–27, July–August 1998.
18. E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September–October 1998.
19. Extensible markup language (xml). http://www.w3.org/XML/, October 2000.
20. Xotcl - extended object tcl. http://nestroy.wi-inf.uni-essen.de/xotcl/, November 2000.

from the Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000), Portland, OR, July 16–19, 2000

Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service

Antonio Carzaniga
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430
USA
carzanig@cs.colorado.edu

David S. Rosenblum
Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
dsr@ics.uci.edu

Alexander L. Wolf
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430
USA
alw@cs.colorado.edu

ABSTRACT

This paper describes the design of SIENA, an Internet-scale event notification middleware service for distributed event-based applications deployed over wide-area networks. SIENA is responsible for *selecting* the notifications that are of interest to clients (as expressed in client subscriptions) and then *delivering* those notifications to the clients via access points. The key design challenge for SIENA is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* of the delivery mechanism. This paper focuses on those aspects of the design of SIENA that fundamentally impact scalability and expressiveness. In particular, we describe SIENA's data model for notifications, the covering relations that formally define the semantics of the data model, the distributed architectures we have studied for SIENA's implementation, and the processing strategies we developed to exploit the covering relations for optimizing the routing of notifications.

1. INTRODUCTION

There is a clear trend among experienced software developers toward designing large-scale distributed systems as assemblies of loosely-coupled autonomous components. A common approach to achieving loose coupling is an *event-based* or *implicit invocation* design style [7]. In an event-based system, component interactions are modeled as asynchronous occurrences of, and responses to, *events*. To inform other components about the occurrences of internal events (such as state changes), components emit *notifications* containing information about the events. Upon receiving notifications, other components can react by performing actions that, in turn, may result in the occurrence of other events and the generation of additional notifications.

Wide-area networks such as the Internet, with their vast number of potential producers and consumers of notifications, create an opportunity for developing novel distributed event-based applications in such fields as market analysis, data mining, indexing, and security. In general, the asynchrony, heterogeneity, and inherent high degree of loose coupling that characterize applications for wide-area networks suggest event interaction as a natural design abstraction for a growing class of distributed systems. Yet to date there has been a lack of sufficiently powerful and scalable middleware infrastructures to support event-based interaction in a wide-area network. We refer to such a middleware infrastructure as an *event notification service* [16].

This paper describes the design of SIENA,[1] an Internet-scale event notification service that is representative of the capabilities we envision for scalable event notification middleware. SIENA is designed to be a ubiquitous service accessible from every site on a wide-area network. As shown in Figure 1, SIENA is implemented as a distributed network of

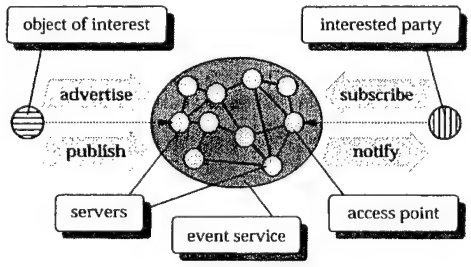


Figure 1: Distributed Event Notification Service.

servers that provide clients with *access points* offering an extended publish/subscribe interface. The clients are of two kinds: *objects of interest*, which are the generators of notifications, and *interested parties*, which are the consumers of notifications; of course, a client can act as both an object of interest and an interested party. Clients use the access points of their local servers to *advertise* the information about notifications that they generate and to *publish* the advertised notifications. Clients also use the access points to *subscribe* for individual notifications or compound patterns

[1] Scalable Internet Event Notification Architectures.

PODC 2000 Portland Oregon

of notifications of interest. SIENA is responsible for *selecting* the notifications that are of interest to clients and then *delivering* those notifications to the clients via the access points.

SIENA is a *best-effort* service in that it does not attempt to prevent race conditions induced by network latency. This is a pragmatic concession to the realities of Internet-scale services, but it means that clients of SIENA must be resilient to such race conditions. For instance, clients must allow for the possibility of receiving a notification for a cancelled subscription. Of course, an implementation would likely adopt techniques such as persistent data structures, transactional updates to the data structures, and reliable communication protocols to enhance the robustness of this best-effort service.

The key design challenge we face in supporting event notification of this kind in a wide-area network is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* of the delivery mechanism. Scalability refers not only to the numbers of publishers and subscribers, and the numbers of notifications and subscriptions, but also to the need to discard many of the assumptions made for local-area networks, such as low latency, abundant bandwidth, homogeneous platforms, continuous and reliable connectivity, and centralized control. Expressiveness refers to the power of the data model that is offered to publishers and subscribers of notifications. Clearly the level of expressiveness influences the algorithms used to route and deliver notifications, and the extent to which those algorithms can be optimized. As the power of the data model increases, so does the complexity of the algorithms. Therefore, the expressiveness of the data model ultimately influences the scalability of the implementation, and hence scalability and expressiveness are two conflicting goals that must be traded off.

While we have not fully explored the nature of this tradeoff, we have investigated a number of carefully chosen points in the tradeoff space. In particular, we designed a data model for SIENA that we believe is sufficiently expressive for a wide range of applications while still allowing sufficient scalability of the delivery mechanism. Based on this data model, we designed distributed server architectures and associated delivery algorithms and processing strategies, and we evaluated and confirmed their scalability. Our description of SIENA in this paper focuses on those aspects of the design that fundamentally impact expressiveness and scalability.

Section 2 presents the data model of notifications in SIENA. Section 3 presents the semantics of SIENA, which is described formally in terms of *covering relations* over advertisements, subscriptions and notifications. Section 4 describes the alternative architectures for SIENA that we have studied and their associated processing strategies; the processing strategies exploit the covering relations for purposes of optimizing the routing of notifications. Section 5 presents an analysis of the complexity of the algorithms, demonstrating the scalability of SIENA with respect to the level of expressiveness we chose for its data model. Section 6 concludes with a discussion of related work, a brief description of a prototype implementation of SIENA, and a discussion of some of our plans for future work.

2. DATA MODEL

As mentioned above, SIENA extends the traditional publish/subscribe protocol with an additional interface function called *advertise*, which is used by an object of interest to inform the event service of the nature of the notifications that it might publish. SIENA also adds the functions *unsubscribe* and *unadvertise* to further inform the event service about the future behavior of interested parties and objects of interest, respectively. Subscriptions can be matched repeatedly until they are cancelled by a call to *unsubscribe*. Advertisements remain in effect until they are cancelled by a call to *unadvertise*.

Underlying SIENA's interface is a *notification data model* (or simply *data model*) that drives the semantics of the service. A notification in the model is an untyped set of typed attributes. For example, the notification shown in Figure 2 represents a stock price change event. Each individual attribute has a *type*, a *name*, and a *value*, but the notification as a whole is purely a structural value derived from its attributes. Attribute names are simply character strings. The attribute types belong to a predefined set of primitive types commonly found in programming languages and database query languages, and for which a fixed set of operators is defined.

string	*class = finance/exchanges/stock*
time	*date = Mar 4 11:43:37 MST 1998*
string	*exchange = NYSE*
string	*symbol = DIS*
float	*prior = 105.25*
float	*change = -4*
float	*earn = 2.04*

Figure 2: Example of a Notification.

The justification for choosing this typing scheme is scalability: Typed notifications, such as one finds for example in the Java Distributed Event Specification [19] and CORBA Notification Service [14], imply a global authority for managing and verifying the type space, something which is clearly not feasible at an Internet scale. On the other hand, we define a restricted set of attribute types from which to construct (arbitrary) notifications. By having this well-defined set, we can perform efficient routing based on the content of notifications. Content-based routing is a powerful technique for selecting and delivering notifications that gives clients the ability to control the precision with which notifications are selected and gives the event service the ability to optimize the processing tasks required for notification delivery. As we discuss in Section 5 and Section 6, content-based routing has distinct advantages over the alternative schemes of channel- and subject-based routing.

In the remainder of this section we discuss two mechanisms for notification selection, namely *filters* and *patterns*, that form the essence of SIENA's extended publish/subscribe protocol. This allows us to fully define the semantics of the interface functions, which we do in Section 3 in terms of what we call *covering relations*. In Section 4 we discuss the use of the covering relations to define the processing strategies

that lead to optimized notification delivery.

An *event filter*, or simply a *filter*, selects event notifications by specifying a set of attributes and constraints on the values of those attributes. Each attribute constraint is a tuple specifying a type, a name, a binary predicate operator, and a value for an attribute. The operators provided by SIENA include all the common equality and ordering relations ($=$, $\neq$, $<$, $>$, etc.) for each of its types, substring ($*$), prefix ($>*$), and suffix ($*<$) operators for strings, and an operator *any* that matches any value.

An attribute $\alpha = (type_\alpha, name_\alpha, value_\alpha)$ matches an attribute constraint $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$ iff $type_\alpha = type_\phi \wedge name_\alpha = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$. We say an attribute α satisfies or *matches* an attribute constraint ϕ with the notation $\phi \sqsubset \alpha$. When α matches ϕ, we also say that ϕ *covers* α. Figure 3 shows a filter that matches price decreases for stock DIS on stock exchange NYSE.

string	*class*	$>*$	*finance/exchanges/*
string	*exchange*	$=$	*NYSE*
string	*symbol*	$=$	*DIS*
float	*change*	$<$	*0*

Figure 3: Example of an Event Filter.

While a filter is matched against a single notification based on the notification's attribute data, a *pattern* is matched against one or more notifications based on both their attribute data and on the combination they form. At its most generic, a pattern might correlate events according to any compound relation. For example, the programmer of a stock market analysis tool might be interested in receiving price change notifications for the stock of one company only if the price of a related stock has changed by a certain amount. Rich languages and logics exist that allow one to express event patterns [12].

We do not attempt to provide a complete pattern language. Our goal is rather to study pattern operators that can be exploited to optimize the selection of notifications within the event service. Here, we restrict a pattern to be syntactically a sequence of filters, $f_1 \cdot f_2 \cdots f_n$, that is matched by a temporally ordered sequence of notifications, each one matching the corresponding filter. An example of a pattern is shown in Figure 4, which matches an increase in the price of stock MSFT followed by a subsequent increase in the price of stock NSCP. In general, we observe that more

string	*what*	$>*$	*finance/exchanges/*
string	*symbol*	$=$	*MSFT*
float	*change*	$>$	*0*

$\bullet$

string	*what*	$>*$	*finance/exchanges/*
string	*symbol*	$=$	*NSCP*
float	*change*	$>$	*0*

Figure 4: Example of an Event Pattern.

sophisticated forms of patterns can always be split into a set of simple subscriptions and then matched externally to SIENA (i.e., at the access point of the subscriber), although this is likely to induce extra network traffic.

Since patterns involve matching of multiple events occurring potentially in different parts of a network, latency effects influence the semantics of the pattern operators. In accordance with SIENA's best-effort semantics, notifications are given timestamps indicating when they were published.[2] This allows the service to detect and account for the effects of latency on the matching of patterns, which means that within certain limits the actual order of notifications can be recognized.

3. COVERING RELATIONS

When a filter is used in a subscription, multiple constraints for the same attribute are interpreted as a conjunction; all such constraints must be matched. Thus, we say that a notification n *matches* a filter f, or equivalently that f *covers* n ($f \sqsubset_S^N n$ for short):

$$f \sqsubset_S^N n \Leftrightarrow \forall \phi \in f : \exists \alpha \in n : \phi \sqsubset \alpha$$

Notice that the notification may contain other attributes that have no correspondents in the filter. Table 1 gives some examples that illustrate the semantics of $\sqsubset_S^N$. The second

subscription		*notification*
string what = alarm	$\sqsubset_S^N$	*string what = alarm* *time date = 02:40:03*
string what = alarm *integer level > 3*	$\not\sqsubset_S^N$	*string what = alarm* *time date = 02:40:03*
string what = alarm *integer level > 3* *integer level < 7*	$\not\sqsubset_S^N$	*string what = alarm* *integer level = 10*
string what = alarm *integer level > 3* *integer level < 7*	$\sqsubset_S^N$	*string what = alarm* *integer level = 5*

Table 1: Examples of $\sqsubset_S^N$.

example is not a match because the notification is missing a value for attribute *level*. The third example is not a match because the constraints specified for attribute *level* in the subscription are not matched by the value for *level* in the notification.

We define the semantics of advertisements with a similar relation $\sqsubset_A^N$. The motivation for advertisements is to inform the event service about which kind of notifications will be generated by which objects of interest, so that it can

[2] With the advent of accurate network time protocols and the existence of the satellite-based Global Positioning System (GPS), it is reasonable to assume the existence of a global clock for creation of these timestamps, and it is hence reasonable for all but the most time-sensitive applications to rely on these timestamps.

best direct the propagation of subscriptions. The idea is that, while a subscription defines the set of interesting notifications for an interested party, an advertisement defines the set of notifications potentially generated by an object of interest. Therefore, the advertisement is relevant to the subscription only if these two sets of notifications have a non-empty intersection.

The relation $\sqsubset_A^N$ defines the set of notifications covered by an advertisement:

$$a \sqsubset_A^N n \Leftrightarrow \forall \alpha_n \in n : \exists \phi_a \in a : \phi_a \sqsubset \alpha_n$$

This expression says that an advertisement covers a notification if and only if it covers each individual attribute in the notification. Note that this is the dual of subscriptions, which define the minimal set of attributes that a notification must contain. In contrast to subscriptions, when a filter is used as an advertisement, then multiple constraints for the same attribute are interpreted as a disjunction rather than as a conjunction; only one of the constraints need be satisfied. Table 2 shows some examples of the relation $\sqsubset_A^N$. The second example is not a match because the attribute

advertisement		*notification*
string what = alarm *string what = login* *string user any*	$\sqsubset_A^N$	*string what = alarm*
string what = alarm *string what = login* *string user any*	$\not\sqsubset_A^N$	*string what = alarm* *time date = 02:40:03*
string what = alarm *string what = login* *string user any*	$\sqsubset_A^N$	*string what = login* *string user = carzanig*
string what = alarm *string what = login* *string user any*	$\not\sqsubset_A^N$	*string what = logout* *string user = carzanig*

Table 2: Examples of $\sqsubset_A^N$.

date of the notification is not defined in the advertisement. The fourth example is not a match because the value of attribute *what* in the notification does not match any of the constraints defined for *what* in the advertisement.

So far we have defined a number of relations that express the semantics of subscriptions and advertisements:

- $\phi \sqsubset \alpha$: attribute α matches attribute constraint ϕ;
- $f \sqsubset_S^N n$: notification n matches filter f, where f is interpreted as a subscription filter;
- $a \sqsubset_A^N n$: notification n matches filter a, where a is interpreted as an advertisement filter;

From these, other relations can be derived:

- $f_1 \sqsubset_S^S f_2$: filter f_1 covers filter f_2, where f_1 and f_2 are interpreted as subscriptions. Formally,

  $$f_1 \sqsubset_S^S f_2 \Leftrightarrow \forall n : f_2 \sqsubset_S^N n \Rightarrow f_1 \sqsubset_S^N n$$

  which means that f_1 covers a superset of the notifications covered by f_2.

- $a_1 \sqsubset_A^A a_2$: filter a_1 covers filter a_2, where a_1 and a_2 are interpreted as advertisements. Formally:

  $$a_1 \sqsubset_A^A a_2 \Leftrightarrow \forall n : a_2 \sqsubset_A^N n \Rightarrow a_1 \sqsubset_A^N n$$

  which means that a_1 covers a superset of the notifications covered by a_2.

The relations $\sqsubset_S^S$ and $\sqsubset_A^A$ can also define the equality relation between filters with its intuitive meaning:

$$f = g \Leftrightarrow g \sqsubset f \wedge f \sqsubset g$$

In the next section we describe how we exploit these derived relations in SIENA's processing strategies.

Unsubscriptions and unadvertisements cancel previous subscriptions and advertisements, respectively. These operations must be performed in the context of the covering relations so that the proper subscriptions and advertisements remain in place. The details of this are complex and are described elsewhere [2, 3].

4. ARCHITECTURES AND PROCESSING STRATEGIES

As shown in Figure 1, the implementation of SIENA comprises a number of interconnected *servers*,[3] each serving some subset of the clients of the service. In effect, SIENA is a wide-area network of pattern matchers and routers overlaid atop some other wide-area communication facility, such as the Internet. One reasonable allocation of such servers might be to place a server at each administrative domain within the low-level, wide-area communication network.

Creating a network of servers to provide a distributed service of any sort gives rise to three critical design issues:

- *Interconnection topology.* In what configuration should the servers be connected?
- *Routing algorithm.* What information should be communicated between the servers to allow the correct and efficient delivery of messages?
- *Processing strategy.* Where in the network, and according to what heuristics, should message data be processed in order to optimize message traffic?

These three design issues have been studied extensively for many years and in many contexts. Our challenge is to find a solution in the particular domain of Internet-scale event notification, leveraging previous results (both positive and negative) wherever possible.

[3]Note that some authors use the term *proxy* or *broker* instead of *server* for this concept.

A pair of interconnected servers use a server/server communication protocol that determines what kinds of information they can exchange, and in which direction. This protocol might make use of any one of a number of lower-level network protocols, such as SMTP or HTTP, through standard encoding and/or tunneling techniques. An interconnection topology and a protocol together define what we refer to as an *architecture* for SIENA. We have studied three basic architectures for SIENA: hierarchical client/server, acyclic peer-to-peer, and general peer-to-peer. We also have studied some hybrids of these three architectures. Because it is not scalable, we ignore the degenerate case of a centralized architecture having a single server.

In the *hierarchical client/server* architecture, the servers form a hierarchical topology, with each server (except the root server) behaving like a SIENA client of the server one level up the hierarchy. As we have demonstrated elsewhere [3], the main problems exhibited by the hierarchical architecture are the potential overloading of servers high in the hierarchy and the fact that each server is a single point of failure. In the *acyclic peer-to-peer* architecture, servers communicate with each other symmetrically as peers in an acyclic undirected graph, adopting a protocol that allows a bi-directional flow of subscriptions, advertisements, and notifications. Removing the constraint of acyclicity from the acyclic peer-to-peer architecture, we obtain the *general peer-to-peer* architecture, which can have multiple paths of bi-directional communication between servers. Allowing redundant connections makes it more robust with respect to failures of single servers. The drawback of having redundant connections is that special algorithms must be implemented to avoid cycles and to choose the best paths. These three basic architectures can be combined to form hybrid architectures, such as an acyclic peer-to-peer topology of subnets, each subnet being a hierarchy.

Once a topology of servers is defined, they must establish appropriate routing paths to ensure that notifications published by an object of interest are correctly delivered to all the interested parties that subscribed for them. In general, we observe that notifications must "meet" subscriptions somewhere in the network so that the notifications can be selected according to the subscriptions and then dispatched to the subscribers. This common principle can be realized according to a spectrum of possible routing algorithms. One possibility is to maintain subscriptions at their access point and to broadcast notifications throughout the whole network; when a notification meets and matches a subscription, the subscriber is immediately notified locally. However, since we expect the number of notifications to far exceed the number of subscriptions or advertisements, this strategy appears to offer the least possible efficiency, and so we consider it no further for SIENA.

To devise more efficient routing algorithms, we employ principles found in IP multicast routing protocols [6]. Similar to these protocols, the main idea behind the routing algorithms of SIENA is to send a notification only towards event servers that have clients that are interested in that notification, possibly using the shortest path. The same principle applies to patterns of notifications as well. More specifically, we formulate two generic principles that become requirements for our routing algorithms:

downstream replication: A notification should be routed in one copy as far as possible and should be replicated only downstream, that is, as close as possible to the parties that are interested in it.

upstream evaluation: Filters are applied and patterns are assembled upstream, that is, as close as possible to the sources of (patterns of) notifications.

These principles are implemented by two classes of routing algorithms, the first of which involves broadcasting subscriptions and the second of which involves broadcasting advertisements:

subscription forwarding: In an implementation that does not use advertisements, the routing paths for notifications are set by subscriptions, which are propagated throughout the network so as to form a tree that connects the subscribers to all the servers in the network. When an object publishes a notification that matches that subscription, the notification is routed towards the subscriber, following the reverse path put in place by the subscription.

advertisement forwarding: In an implementation that uses advertisements, it is safe to send a subscription only towards those objects of interest that intend to generate notifications relevant to that subscription. Thus, advertisements set the paths for subscriptions, which in turn set the paths for notifications. Every advertisement is propagated through the network, thereby forming a tree that reaches every server. When a server receives a subscription, it propagates the subscription in reverse, along the path to the advertiser, thereby *activating* that path. Notifications are then forwarded only through the activated paths.

In the simplest implementation of the subscription forwarding (advertisement forwarding) algorithms, all subscriptions (advertisements) would be stored at all servers. However, we can optimize their implementation by exploiting the covering relations defined in Section 3. When a subscription reaches a server (either from a client or from another server), the server propagates that subscription only if it defines new selectable notifications that are not in the set of selectable notifications defined by any previously propagated subscription. Three benefits accrue from this approach: First, we reduce network costs by pruning the propagation of new subscriptions. Second, we reduce the storage requirements for servers. Third, by reducing the number of subscriptions held at each server, we reduce the computational resources needed to match notifications at that server. We use a similar strategy for propagation of advertisements.

In order to keep track of the propagation of subscriptions (and similarly advertisements), every server maintains a partially ordered set (*poset*) of subscriptions, where the partial order relation is defined by the covering relations. Each server associates with each subscription s a set of subscribers

subscribers(s) and a set of neighbor servers *fwds*(s) to which s has been forwarded. *fwds*(s) is a subset of *neighbors*, the set of all neighbors of the server in the network.

When a server receives a subscription f from a client or a neighbor server U, it looks in its subscriptions poset P_S for either

1. a subscription f' that covers f and has U among its subscribers: $f' \sqsubset_S^S f \wedge U \in$ *subscribers*(f'). In this case, the procedure that handles the subscription returns with no effect; or
2. a subscription f' that is equal to f and does not contain U in its subscribers: $f' \sqsubset_S^S f \wedge f \sqsubset_S^S f'$. Here the server adds U to *subscribers*(f'); or
3. two possibly empty sets $\overline{f}$ and $\underline{f}$, representing the immediate predecessors and the immediate successors of f respectively. Here the server inserts f as a new subscription between $\overline{f}$ and $\underline{f}$, and adds U to *subscribers*(f).

In cases 2 and 3, the server also removes U from all the subscriptions in P_S that are covered by f, and then removes from P_S those subscriptions that have no other subscribers.

Next, the server forwards the subscription to some of its neighbors. Formally, given a subscription f in P_S, let *fwds*(f) be defined as follows:

$$fwds(f) = neighbors - NST(f) - \bigcup_{f' \in P_S \wedge f' \sqsubset_S^S f} fwds(f')$$

In other words, f is forwarded to all neighbors except those not on any spanning tree rooted at an original subscriber of f (the second term in the formula), and those to which subscriptions f' covering f have been forwarded already by this server (the last term in the formula).

The last term in the formula represents the optimization that the server can make in the situation where more generic subscriptions have been propagated already to some neighbors.

Hence, in the process of forwarding subscriptions or advertisements, SIENA exploits commonalities among subscriptions or advertisements. In practice, SIENA prunes the propagation trees by following only those paths that have not been covered by previous requests. The derived covering relations $\sqsubset_S^S$ and $\sqsubset_A^A$ defined in Section 3 are used to determine whether a new subscription or a new advertisement is covered by a previous one that has already been forwarded.

Unsubscriptions and unadvertisements are handled in a similar way to undo the effect of the affected subscriptions or advertisements.

To match patterns, servers assemble sequences of notifications from small subsequences or from single notifications according to the advertised paths along which notifications will be propagated. For this reason, advertisement forwarding algorithms are necessary to implement the *upstream evaluation* principle for event patterns.

The interested reader is referred to our other publications [2, 3] for details on how we apply these processing strategies to the different architectures.

5. ANALYSIS

Consider two extremes of expressiveness. In a *channel-based* model of event notification, notifications are fed into what amounts to a discrete communications pipe. Subscriptions are made by simply identifying the pipe (i.e., channel) from which notifications are expected to flow; the notion of "filtering" then reduces to channel selection. Since the contents of notifications are not used in routing, it is not necessary to define any service-visible structure within notifications. The covering relations become an equality check on the identifier of the channel, thus making the routing of notifications very efficient. However, the resulting notification selection mechanism is simplistic, and too weak for some applications.

At the opposite extreme, the structure of notifications, the types of attributes within notifications, and the operators that can be applied to those attributes are all application defined, perhaps employing the full expressive power of a Turing-complete language. However, the operators, which are used by the service to perform notification selection, would then be of an arbitrary, unknown, and potentially unbounded complexity. Moreover, the computation of the covering relations that allow the pruning of propagation trees, such as $\sqsubset_S^S$, might be undecidable.

These considerations led us to a level of expressiveness in SIENA at which notification structure, attribute types, and attribute operators approximate those of the well-understood and widely-used database query language SQL. In particular, SIENA supports the definition of filters that essentially implement a significant subset of the SQL *select* query.

The covering relations are well behaved and predictable in the sense that they exhibit an arguably reasonable computational complexity deriving from the expressiveness of filters: Assuming a brute-force and unoptimized algorithm, the complexity of computing $\sqsubset_S^N$ on a given subscription and a given notification is $O(n+m)$, where n is the number of attribute constraints in the subscription filter and m is the number of attributes in the notification. The complexity of each individual comparison is $O(1)$ for all the predefined types we have included in SIENA. The only exception is for the string type, but efficient comparison algorithms are well known. The complexity of computing $\sqsubset_S^N$ reflects the computation of an intersection between the attribute values in a notification and constraints on those values appearing in a subscription.

The complexities of computing $\sqsubset_S^S$ and $\sqsubset_A^A$ are all $O(nm)$, where n and m represent the number of attribute constraints appearing in the respective subscription and/or advertisement filters. This complexity represents a comparison between each attribute constraint in one filter and any corresponding attribute constraints in the other filter. Checking a covering relation between filters amounts to a universal quantification. But given our choice of types and operators, comparing a pair of attribute constraints can be reduced to evaluating an appropriate predicate on the two constant val-

ues of the constraints, with a complexity $O(1)$. For example, to see if $[x > k_1]$ covers $[x > k_2]$ we can simply verify that $k_2 \geq k_1$.

We also restricted the expressiveness of patterns in SIENA in the interests of efficiency. As we discuss in Section 2, patterns are simple sequences of filters. The computational complexity of recognizing a pattern is $O(l(n+m))$, where l is the length of the pattern. This means that it is linear in the number of filters, whose covering relation $\sqsubset_S^N$ has complexity $O(n+m)$.

Our conclusion from this analysis is that the optimizations presented in Section 4 are effective, since they derive from the reasonable complexity of the covering relation computations. In fact, the factors n and m are, in practice, likely to be relatively small (typically less than 10), making the computations negligible compared to the network costs they are attempting to reduce. This is all achieved with an expressiveness that approximates SQL.

With respect to the scalability of the service across a distributed network, we have carried out simulation studies to determine how the architectures and processing strategies perform over an extensive range of scenarios having different network configurations and application behaviors. Details of the simulation framework and specific results are available elsewhere [2, 3].

6. CONCLUSION

In this section we briefly review related work in event notification services and discuss our prototype implementation of SIENA. A more complete discussion of these topics is presented elsewhere [2, 3].

We can compare related technologies from the perspective of their server architecture, which affects scalability, and from the perspective of their subscription language, which determines expressiveness. Table 3 presents such a comparison in terms of the architectures described in Section 4 and in terms of a classification of subscription languages shown in Table 4.

We classify subscription languages based on their *scope* and *expressive power*. Scope has two aspects: (1) whether a subscription is limited to considering a single notification (thus reducing the language to that of filters) or whether it can consider multiple notifications (thus involving both filters and patterns); and (2) whether a subscription is limited to considering a single, designated field in a notification or whether it can consider multiple fields. Expressive power is concerned with the sophistication of operators that can be used in forming subscription predicates, ranging from a simple equality predicate to expressions involving only predefined operators to expressions involving user-defined operators. As we point out in Section 5, user-defined operators suffer from the disadvantage of having arbitrary, unknown, and potentially unbounded complexity. In fact, we have observed that subscription languages with user-defined predicates are rare; in Table 3 we have combined the language classes corresponding to predefined and user-defined predicates because only a single entry, object-oriented active databases, makes use of user-defined predicates. Notice that a property of the classification in Table 4 is that the classes are inclusive: channel-based languages are a special case of subject-based languages, which in turn are a special case of content-based languages, and so on. Therefore, a technology appearing in a given row of Table 3 implicitly offers the subscription language of that row and any rows above it in the table.

We have implemented a prototype of SIENA[4] and used it as the event service of an agent-based, wide-area software deployment system called the Software Dock [8]. The current implementation of SIENA offers two APIs, one for C++ and the other for Java. Both interfaces support the data model and subscription language described in Section 2. Two event servers are also provided by the current implementation. One (written in Java) is based on the hierarchical client/server algorithm, while the other one (written in C++) is based on the acyclic peer-to-peer architecture with the subscription forwarding algorithm. These two types of servers have been used together (thus forming a hybrid topology) in the Software Dock.

Our future work will continue exploring the interplay of expressiveness and scalability, as well as other important issues concerning Internet-scale event notification such as wireless networking, application frameworks, and security. Security is a particularly intriguing issue. While traditional security techniques based on encryption can be employed in an event notification service to achieve certain kinds of security, such as authentication of publishers and integrity of notifications, the very nature of the service requires a rethinking of basic principles of secure communication. For instance, traditional notions of privacy do not apply because senders (i.e., publishers) do not designate the intended recipients (i.e, subscribers) of their messages (i.e., event notifications). Nevertheless, privacy considerations may apply in new ways, such as to prevent one client from having access to another client's subscriptions.

Acknowledgments

We would like to thank Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, and André van der Hoek for their considerable contributions in discussing and shaping many of the ideas presented in this paper.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0253, F30602-97-2-0021, F30602-98-2-0163 and F30602-99-C-0174; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Researach

[4] `http://www.cs.colorado.edu/serl/dot/siena.html`.

Subscription Language	Architecture: *centralized*	Architecture: *hierarchical client/server*	Architecture: *general peer-to-peer*
channel-based	Field [15] CORBA Event Service [13] Java Dist. Event Spec. [19]	CORBA Event Service [13]	IP multicast [6] iBus [18]
subject-based	ToolTalk [9]	NNTP [10] JEDI [5] TIB/Rendezvous [20]	*(none known)*
content-based	Elvin [17]	Keryx [21] Yu et al. [22]	Gryphon [1]
content-based with patterns	GEM [12] Yeast [11] CORBA Notification Service [14] object-oriented active databases [4]	SIENA	SIENA

Table 3: A Classification of Related Technologies.

Power	Scope: *Single Notification Single Field*	Scope: *Single Notification Multiple Fields*	Scope: *Multiple Notifications Multiple Fields*
Simple Equality	channel-based	—	—
Expressions with Predefined Operators	restricted subject-based	restricted content-based	restricted content-based with patterns
Expressions with User-defined Operators	general subject-based	general content-based	general content-based with patterns

Table 4: Typical Features of Subscription Languages.

Laboratory or the U.S. Government.

7. REFERENCES

[1] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, TX USA, May 1999.

[2] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, Oct. 1999.

[4] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Mateo, 1996.

[5] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, To appear.

[6] S. E. Deering and D. R. Cheriton. Multicast routing in datagram networks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–111, May 1990.

[7] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of VDM '91: 4th International Symposium of VDM Europe on Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, Oct. 1991. Springer–Verlag.

[8] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore MD, U.S.A., May 1997.

[9] A. M. Julienne and B. Holtz. *ToolTalk and open protocols, inter-application communication.* Prentice–Hall, Englewood Cliffs, New Jersey, 1994.

[10] B. Kantor and P. Lapsley. Network news transfer protocol—a proposed standard for the stream-based transmission of news. Internet Requests For Comments (RFC) 977, Feb. 1986.

[11] B. Krishnamurthy and D. S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, Oct. 1995.

[12] M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[13] Object Management Group. CORBAservices: Common object service specification. Technical report, Object Management Group, July 1998.

[14] Object Management Group. Notification service. Technical report, Object Management Group, Nov. 1998.

[15] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[16] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 344–360. Springer–Verlag, 1997.

[17] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Queensland, Australia, Sept. 3–5 1997.

[18] SoftWired AG, Zurich, Switzerland. *iBus Programmer's Manual*, Nov. 1998. http://www.softwired.ch/ibus.htm.

[19] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Java Distributed Event Specification*, 1998.

[20] TIBCO Inc. Rendezvous information bus. http://www.rv.tibco.com/rvwhitepaper.html, 1996.

[21] M. Wray and R. Hawkes. Distributed virtual environments and VRML: an event-based architecture. In *Proceedings of the Seventh International WWW Conference (WWW7)*, Brisbane, Australia, 1998.

[22] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for Internet-scale event services. In *Proceedings of WETICE '99*, Stanford, CA, June 1999.

WREN—An Environment for Component-Based Development

Chris Lüer, David S. Rosenblum

Department of Information and Computer Science
University of California, Irvine, CA 92697

Technical Report #00-28

September 1, 2000

Abstract

Prior research in software environments focused on three important problems—tool integration, artifact management, and process guidance. The context for that research, and hence the orientation of the resulting environments, was a traditional model of development in which an application is developed completely from scratch by a single organization. A notable characteristic of component-based development is its emphasis on integrating independently developed components produced by multiple organizations. Thus, while component-based development can benefit from the capabilities of previous generations of environments, its special nature induces requirements for new capabilities not found in previous environments.

This paper is concerned with the design of *component-based development environments*, or CBDEs. We identify seven important requirements for CBDEs and discuss their rationale, and we describe a prototype environment that we are building to implement these requirements and to further evaluate and study the role of environment technology in component-based development. Important capabilities of the environment include the ability to locate potential components of interest from component distribution sites, to evaluate the identified components for suitability to an application, to incorporate selected components into application design models, and to physically integrate selected components into the application.

1 INTRODUCTION

It has been stated that component technology, while successful in industry, has not received the attention it deserves from the research community [12]. Industrial component models are still rudimentary, and the approaches of different vendors vary strongly. Research is necessary in order to define a common foundation of component technology, and to identify areas in which current standards and tools have to be extended.

Software environments are one area that can benefit especially well from further research. Software development environments (SDEs) were originally designed to integrate collections of tools and to manage locally created development artifacts. Later, process centered software engineering environments (PSEEs) were developed to facilitate the use of well-defined processes to guide development. In order to provide tool integration and process-based guidance for the special needs of component-based development, we envision a new generation of environments, *component-based development environments*, or *CBDEs*. Reusable components developed by and licensed from other organizations cannot be treated in the same way as artifacts that were developed in-house, since it is usually not possible to change their implementations. Therefore, new approaches are needed to support identification, retrieval and integration of such components within an environment in an Internet-scalable way.

Szyperski defines a component as follows [21]:

- *A unit of independent deployment.* This means that a key goal of component technology is to facilitate code reuse [10]. A component is a piece of code that has been prepared for reuse. This is opposed to code scavenging, where code that was not explicitly intended to be reusable is being reused. Though initially more expensive, we view design-for-reuse as being the superior approach to enabling reuse.
- *A unit of third-party composition.* Reuse will pay off only when reusing a component that was developed by another organization is significantly easier than redeveloping it. In the ideal case, an application would be composable from components by domain experts without actual programming.
- *Without persistent state.* A component is a piece of code, or a set of abstract data types. In an object-oriented system, a component is a set of classes. A component is not an object or a set of objects.

A CBDE must provide its users with information about components. The users have not designed the components themselves, so they depend on the environment to learn about them. With the use of components, the focus of tools shifts from implementation to design, since the goal of component reuse is to minimize implementation effort. The user has to decide which components fit best into the envisaged architecture, so the environment should be able to visualize the dependencies among the components. Because components are developed by third parties, the environment should provide the means to access components located at remote sources.

In this paper, we present requirements for CBDEs, and we describe a prototypical environment, WREN, which we are building based on these requirements. Our prototype is based on the Java language and the Java Beans component model. Components packaged as described in this paper are backwards compatible with Java Beans, although they have been extended in various ways.

In Section 2, we describe seven requirements we believe to be fundamental for the design of CBDEs, and we discuss the rationale for these requirements. In Section 3, we present WREN, a prototypical implementation of such an environment. Sections 4 and 5 discuss related work and our conclusions, respectively.

2 REQUIREMENTS OF COMPONENT-BASED DEVELOPMENT ENVIRONMENTS

While a number of specialized technologies have been produced in both research and industry to facilitate particular aspects of component programming and reuse, we are unaware of any attempts to provide comprehensive, integrated environment support for the full range of lifecycle activities that must be undertaken in component-based development. In this section we identify seven requirements for CBDEs that address the needs of component-based development. Some of these requirements are addressed by industrial component models, while some of them are not yet widely adopted and are perhaps even controversial. Briefly,

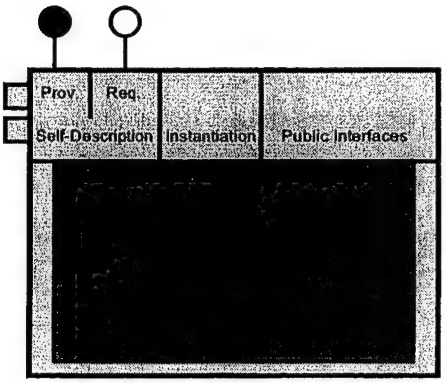


Figure 1: Structure of a Component. Public parts are light grey and private parts are dark grey.

- Accepted rules of *modular design* should be followed. The environment should support a separation between the private and public parts of a component.
- The environment should support and exploit component *self-description*, meta-information that is stored directly inside of the component. It is used in a limited way in industrial component models like Java Beans and COM.
- Components should be defined and accessed within a *global namespace of interfaces*, which provides a method to name interfaces in a globally (worldwide) unique way. This reduces the problem of semantics matching to namespace agreement.
- The environment should support a bipartite development process comprising two parts: *component development* and *application composition*. The former deals more with technical issues of individual component development, while the latter is more application-oriented.
- Application composition consists of configuration of the components and the design and implementation of additional functionality that is not available in components. The environment should support two methods of configuration: *connection* and *adaptation*.
- A CBDE should support *multiple views*, including a development view and a composition view to represent the two halves of the component-oriented process, and a type view and an instance view to show different aspects of the composition view, using an explicit architectural model to represent the overall structure of the application.
- The maintenance problems associated with component technology should be addressed by the environment through *reuse by reference*.

We next discuss the rationale for these requirements.

Modular Design

Figure 1 presents a generic model of a component that has been prepared for use in a CBDE. A component should be divided into a public part and a private part according to the principle of information hiding or encapsulation [17]. The private part is not accessible from the outside; it contains implementations (in the form of classes) and resources (for example, graphics or help files). The public part contains the self-description of the component, an instantiation mechanism, and optionally public interface definitions. The instantiation mechanism is necessary so that clients can retrieve instances of the data types implemented by the component. To do so, a client specifies only the interface of the data type of which it wants to retrieve a new instance. The decision of which actual class is used to provide this instance is hidden and made by the component itself. Public interface definitions are interfaces that are contained in the component and made accessible to other components, which might want to implement them. The purpose of the self-description and the *provides* and *requires* ports is described below.

The basic unit of syntactical description is the interface. An interface is a named set of operations that describes an abstract data type. Explicit interfaces make it possible to provide alternative implementations

(in the form of classes) to a given type (specified through an interface). Thus, if we ensure that components use only interfaces for their specification, the actual implementations will be encapsulated and exchangeable. Interfaces can be specified independently from the components that implement them so that competing manufacturers can offer components that are interchangeable.

Self-Description

Self-description is a central idea of component technology. Components should be able to provide information about themselves in a systematic way to a CBDE, and to other components a runtime. Description that is contained in the component itself has many advantages over externally stored description. External description, such as documentation stored in text files, can get lost, often has to be updated manually, and cannot easily be queried by development environments. On the other hand, many forms of self-description can be automatically generated and embedded within the component implementation.

The self-description of a component should contain all the information that is needed to reuse it. This is, first, information about the services that the component provides, and second, information about the services the component requires to work. The information in both of these categories can be categorized into five levels [1]:[1]

1. *Syntactic Level:* This level describes the signatures of the abstract data types that are provided or required. Self-description at the syntactic level is a common feature of many component technologies.
2. *Behavioral Level:* This is a level for informal, semi-formal or formal semantic description of data types.
3. *Synchronization Level:* This level is used to enable cooperating components to agree about concurrency issues.
4. *Quality-of-Service Level*: At this level is self-description regarding all non-functional requirements of the component, such as response latency, precision of results, and memory requirements.
5. *Non-Technical Level:* This is a level for business-oriented information, such as price, contact address for support, quality certifications, and so on.

To different degrees, all of these levels can participate in negotiations regarding the level of service a component delivers to an application. The services offered or requested need not be static; they can be dynamically adapted to conditions of the environment.

Providing all this information in the component itself instead of in the form of documentation that is stored elsewhere makes the information available to composition tools. A composition tool can check if two components can be connected without having access to their source code, by querying the self-description. In a similar way, component repositories can leverage component self-description for searching and retrieving components. They can check a user's requirements against the self-description. A component self-description standard could reduce the need for a repository standard, because component repositories could then be very simple when all the information about components is stored where it belongs—in the components.

In a similar way, configuration management can be simplified by the use of self-describing components. Typically, configuration management tools store external information about the dependencies between components. This is necessary when arbitrary files are managed. The task becomes easier, however, when the application is built out of self-describing components. Self-description moves dependency information into the components, where it is encapsulated so that it can easily evolve with the evolution of the component implementation.

Global Namespace of Interfaces

A global namespace of interfaces partly solves the problem of how a CBDE will ensure consistency between the semantics of a provided component to the semantics required of the component; Zaremski and Wing have studied this problem in the context of *signature matching* [26]. While there may be different interfaces providing the same functionality, in a global namespace of interfaces, two interfaces with the same name are intended to be functionally equivalent. On a fundamental level, this greatly simplifies the problem of matching provided components to required semantics, since the problem is reduced to name equality. Only when components do not match at the interface level is human intervention required: Either they are truly incompatible (i.e., incompatible on a semantic level), or the incompatibility is only syntactic,

[1] The fifth level described in this list is a level we have added to the classification of Beugnard et al.

so that they can be matched by simple manual adaptation (for example by wrapping one of them). Of course, mechanisms are still needed to ensure that a component correctly implements the semantics promised by its interfaces, but this problem already existed alongside the component matching problem.

Component Development and Application Composition Processes

A component-oriented development process looks different from a traditional one. The process is bipartite: The development of components, and the composition of an application from the components are separated. Typically, the two process parts will be executed by different organizations, the component manufacturer and the organization that wants to license and reuse the manufactured components. We refer to these organizations abstractly as the *component developer* and the *application composer*.

Component development is a traditional development process since all the usual lifecycle phases are traversed. The main difference is that the end product is not a complete application. This means that the product is comparatively small, which may make development processes suited to small projects preferable. Often, the component might not have a user interface of its own, but will be required to interact with a GUI through a standardized interface instead. Components are to be used in unknown contexts; this makes quality management essential. An isolated component cannot be beta tested, so correctness has to be assured by other means, such as internal testing and code inspection. In this way, component development has a certain similarity with the development of embedded systems. The "shape" of a component will determine the architecture of systems reusing this component. Therefore, the component developer should make sure that a component works well together with related components and can be fit easily into an architecture.

A CBDE can support traditional component development, but it must excel at supporting application composition, which should focus on the business aspects of an application. In the ideal extreme, all components can be *bought* or otherwise obtained. The application composer must select the right components, connect and adapt them, and identify components that might be missing. The goal of component reuse is to minimize the implementation phase of an application. Instead of spending effort on programming, reusable components are bought. In the near future, it will not be possible to completely eliminate the implementation phase except for trivial projects, but it can be minimized and simplified using appropriate components and environment capabilities.

The application composition process already differs from a regular process in the requirements phase. In requirements, and even more so in design, the component market must be taken into consideration. Finding components that match arbitrary requirements will be difficult or impossible. The cost savings gained by component reuse will often make it feasible to adapt requirements and design to the components that are available. Thus, the availability of components must be accounted for during the whole process.

Connection and Adaptation

Once the decision to reuse a certain component is made, it will have to be configured within a CBDE. Component configuration consists of connection and adaptation. Components have to be connected with each other so that they can cooperate. In the simplest case, the connector is just a link between a given required service and a given provided service. In other words, a connector establishes how a requirement is fulfilled. But connectors can be more complex; it is useful to have them encapsulate functionality that logically belongs within a shared infrastructure (for example, communication protocols in a distributed system) rather than to either of the two components that are being connected [20] [5].

Adaptation increases the value of components [2]. The more flexible and adaptable a component is, the more often it will be reused. Ideally, a component will provide ways for application composers to adapt it; popular adaptation methods include wizards and property sheets, which support internal adaptation. However, a component manufacturer will not be able to foresee all adaptations that might be necessary. For this reason, there should be means to adapt a component without having to interact with it, through external adaptation. One way to do so is to implement a wrapper component that maps the interface of a component to a different interface. Another solution is an adaptation connector, which is specifically written to make interoperation between two components possible. Unfortunately, external adaptation has a limited scope, because the internals of the component that is adapted are hidden. Usually, external adaptation is used to convert between interfaces that approximately have the same semantics, but use a different syntax. In this case, a wrapper can be implemented very easily by a human, though it cannot be generated automatically.

Multiple Views: Development View and Composition View

CBDEs should aide both the viewpoint of the component developer and the viewpoint of the application composer. Although a component developer will not necessarily compose any application, the application

composer will have to develop some components that are specific to the application being built. So, the application composer may have to switch between both roles.

The component development view of a CBDE will look very much like a traditional, non-component-oriented environment, including code editor, compiler, debugger, and so on. But it should provide a way to distinguish the public features of a component from its internal, private features. In many languages this is done through corresponding keywords. A specific graphic design notation that shows the outside (the specification) versus the inside (the implementation) is helpful. Further, the code for instantiation and syntactic self-description can easily be generated by the tool from a graphical representation, such as a UML class diagram.

The application composition view will be less traditional. Most importantly, it abstracts from the hidden internals of the components. Even if a component was written by the composer, and so its internals are accessible, that part should be hidden. Since the purpose of component technology is to minimize the implementation effort, the composition view will look very much like a design view.

Multiple Views: Instance and Type View

The composition view should be divided into two subviews. The type view will show the components that are used and their dependencies. The instance view will show selected instances of some of the data types provided by the components, and how they are configured.

Instance views are known from commercial development environments (for example, Web Gain Visual Café, or IBM Visual Age). They allow the composer to visually adapt and connect certain objects (instances of classes), such as GUI elements in dialogs, menus and so on. A typical example of a connection type supported by instance views is an attribute-to-attribute connection: Each time one attribute changes, the other one is automatically updated, so that they are always equal. Graphical instance views save implementation effort by providing a way to specify trivial code in a visual manner. Unfortunately, their applicability is limited. There is no way to specify dynamic behavior in them, such as instantiation. Objects that cannot be created at program initialization, but only later, cannot be represented. For this reason, instance diagrams are best suited to show objects that are singletons, such as unique GUI dialogs, or a database. They are less suited for objects that represent business logic or container data structures.

Type views are on the same logical level as UML class diagrams, but instead of classes, components are shown, and instead of associations or inheritance relations, connectors are shown. The purpose of the type view is to show how the various components depend on each other, which components are used in the application, which might be exchanged, and what might be missing. The composer has to be able to see what each component provides and requires, for example in order to identify requirements that are not yet met.

The type view shows the architecture of the application that is being composed, and serves as a basis for design decisions. For example, once a need is identified, the composer will have to search in a *component market* for components that fulfill this need. Typically, more than one such component will be available. The composer can use the type view to check which of them best fits into the architecture, and then this can be used a selection criterion together with aspects like quality of service or price.

Multiple Views: Explicit Architectural Diagrams

UML component diagrams cannot be used to show unmet requirements since they provide no syntactic notation for entities that are required to exist but do not. For this reason, we propose *provides* and *requires* ports as a diagrammatic notation. The concept of ports is known from, among others, the architecture description language Darwin [11], and they are also used in UML for Real-Time UML [19]. A port is a part of a component that is expected to be linked to another port with a connector, but is not necessarily connected at all times. Each port is either a *requires* port or a *provides* port, and connectors are directed from *requires* to *provides*, so that they can be interpreted as use-relations. A port that is not connected thus shows that something is missing; the component is not yet ready to be used. In this way, an application composer can keep track of the completeness of the application that is being built by watching the status of the ports.

The need for explicit ports shows that reused components developed by other organizations (off-the-shelf components) have to be treated differently from components that were developed within the project at hand. While newly developed components can be modified whenever necessary, and new dependencies can be added, reused components are typically bought without source code, and so they cannot be modified beyond what is possible through adaptation. With reused components, the ports will be fixed and unchangeable. Even if reused components were developed in-house (and their source is available) the

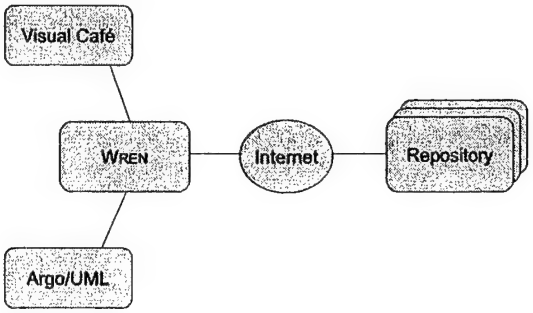


Figure 2: Architecture of WREN.

learning cost may make changing their private implementation prohibitive. As a consequence, the structures of reused components have to be considered as fixed requirements in the software process.

Reuse by Reference

Component reuse exacerbates the problem of maintenance. An application that consists of a large number of independently bought components will be much harder to update than a traditional, monolithic application, since each component will have individual updates from its manufacturer. *Reuse by reference* is a possible solution to this problem.

Reuse by reference means that a single, worldwide master copy of a component is referenced over the Internet. Copying is performed by the CBDE only in the form of caching for performance purposes. A permanent connection is established by the CBDE between the client application that uses the component and the repository on which the master copy resides. In this way, the component can be updated automatically.

3 THE WREN PROJECT

WREN[2] is a prototypical implementation of an integrated CBDE that we are building to realize and evaluate the requirements discussed in Section 2. Figure 2 depicts the architecture of WREN. As the figure shows, the CBDE is integrated with Argo/UML [18], a UML design tool, and Web Gain Visual Café, a software development environment. The CBDE is a client of one or more component repository servers; we have built such a server, which
communicates with the CBDE through a simple protocol that runs on top of TCP/IP.
In the following, we describe the features of WREN, its use for application composition and how it interacts with the other applications. Then we discuss some issues of the design of the environment. Support for component development is planned, but not yet implemented except as supported in Visual Café.

Programming Language

We chose Java as the programming language for WREN because it supports component technology and addresses our requirements for CBDEs in multiple ways:

- It supports encapsulation through its access modifiers. Java provides encapsulation on two levels, class and package. Since components can contain more than one class, we use the package-level access modifiers to implement components.
- In Java, signature descriptions can be obtained at runtime through the reflection mechanism of the language. This makes it possible to automatically generate component self-descriptions and simplifies component configuration.
- Java supports interfaces as explicit entities similar to classes. This has the advantage that interfaces and classes can be treated uniformly. A component can provide both classes (i.e. implementations of interfaces) and interfaces.
- Java interfaces reside in a global, worldwide namespace, which is created through the naming convention for package names used in Java: A name should start with the reversed Internet domain

[2] Sir Christopher Wren (1632–1723) is remembered for his designs of 51 churches rebuilt in London after the Great Fire of 1666. Each design was unique but was a recognizable variant of an elegant new architectural style.

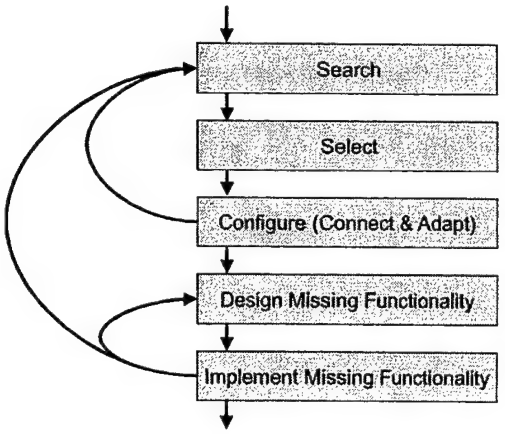


Figure 3: Application Composition Process.

name of the manufacturing organization. For example, an interface for abstract data type `foo` developed at the University of California, Irvine, could be named `EDU.uci.foo`.

- Java supports dynamic linking and late binding. This makes it possible for a CBDE to configure a component application without need for additional external tools or interprocess communication.

Application Composition Process

Figure 3 summarizes the application composition process that is facilitated by WREN. While the process is currently not enforced in any way, the environment is designed to support each part of this process. After the requirements are identified, relevant components have to be found. Repositories should be *searched* in a top-down manner; once the most important components are identified, it will be easier to formulate search criteria for the rest. A typical search will produce far more candidates than needed, many of which will be mismatches. So, in the next step, the composer has to *select* among the found components. All levels of component self-description will be used in this activity. Components that have been selected need to be *configured* (connected and adapted). Now, missing components, which are required by the selected components, have to be found and integrated, so the process loops back to the search step. Unlike the beginning of the process, where components can be searched for only by vague, natural language criteria, the interfaces specified by the *requires* ports can now be used to automatically search for compatible components. There will still be multiple matches, so that the composer will have to select again according to soft criteria such as quality of service. After several iterations, all components that can be reused will have been found and configured. Missing functionality for which no components can be found will have to be designed and implemented in a traditional manner.

In summary, application composition is an iterative process involving searching, selecting, and configuring components. Searching can be automated in part, but selection and configuration are creative tasks that require design experience. As a result of these three steps, there are three sets of components that exist during the process. First, there are *available components*, which are all components that match the search criteria. Out of these, the composer has to select those that are to be used, the *selected components*. Given the set of selected components, the environment can identify *missing components*. These are all the implementations that are required by one of the selected components but not fulfilled by another one. Missing components can only be described in the form of incomplete requirements, since they are not found yet.

Searching for Components

Typically, the application composer will start with a broad search for natural-language keywords. The composer enters the search terms into the CBDE, which in turn sends a search command to all the repository servers it knows about.[3]

Search commands are implemented as pieces of mobile code. The repository server executes the mobile code and allows it to search through all its stored components. The mobile code then queries the self-

[3] Space considerations prevent us from using screenshots to illustrate the environment's search and select features.

description of the components in order to check them against some associated search criteria. The default search command just checks the search terms against a list of keywords provided by the semantic self-description of a component. The repository architecture leaves the decision of how to search to the client CBDEs, however. A CBDE could easily replace this basic search strategy with a more complex one, for example one that makes use of natural language processing features. The use of mobile code for searching the repository makes the repository itself an almost trivial piece of software. All the management of meta-information, dependencies, and so on that is typically done by a reuse repository is delegated to the components themselves, or rather their self-description.

When a component is found that matches the search criteria, a stub is transferred to the client. The stub contains the self-description information and can handle calls to the implementation part of the component. The CBDE adds the component to its set of available components, and uses the component stub to present information about the component to the composer.

Component Selection

Often, the set of available components will be very large, since it is hard to specify search criteria in a sufficiently precise way. The application composer then uses the environment to browse through the available components, to look at their properties, and to select the ones that are needed.

WREN has a window that displays a selection of relevant properties of the available components for easy comparison. Among them are name, manufacturer, size, price, and number of *provides* and *requires* ports. The numbers of ports allow an easy estimation of the architectural complexity of the component. For example, a component that has zero *requires* ports will be at the bottom of the architecture because it does not depend on any other components. An alternate view of the available components is sorted by the interfaces that the components implement, making it easy to compare all components that are possible suppliers for a given data type. However, since a component usually implements more than one interface, this view is less compact.

From the requirements of the selected components, the environment generates the set of missing components. The environment checks through the *requires* ports and adds an entry to the set of missing components for each required data type that is not provided by any of the selected components. It may be possible that several of the missing data types are implemented by one component, so the size of this set does not permit conclusions about the number of actual components that have to be found.

Now, the "find missing components" feature of the environment can be used to automatically search the repositories for all matching components. It is possible that more than one component matches a requirement for a "missing component", so that the composer will have to select among them. The process of searching and selecting components has to be repeated until the set of missing components is empty or the composer decides to reimplement the missing components. To do so, a missing component can be marked as "self-implemented"; this will exclude it from further searches.

Type-Oriented Component Configuration

As shown in figure 4, the CBDE has a diagram editor that allows the composer to connect components. The CBDE uses Argo/UML, an open-source design environment, to display UML component diagrams that are augmented by ports as discussed in Section 2. The selected components are represented in these diagrams by icons provided in the component's self-description. When the diagram is opened, all selected components are displayed with their respective *requires* and *provides* ports. *Requires* ports are depicted as hollow circles, *provides* ports as filled circles. Each port is labeled with the name of the interface for which an implementation is required or provided. The composer can drag the components and can create directed connections in the form of UML dependencies from *requires* ports to matching *provides* ports. Each *provides* port can be used by any number of *requires* ports, but a *requires* port cannot be connected to more than one *provides* port. It is not possible to change the number or names of the ports of a component, since this would require access to its source code.

A component diagram in this style gives an overview of the architecture that is being built and makes it easy to see which requirements are not yet fulfilled. Each unfulfilled requirement corresponds to a *requires* port that is not connected to any *provides* port. Figure 4 provides an example of this with `DisplayBean`'s *requires* port `Printer`. In a similar way, one can see which components may be affected when one component is exchanged for a compatible one.

Component adaptation as described in Section 2 is not yet implemented in the prototype.

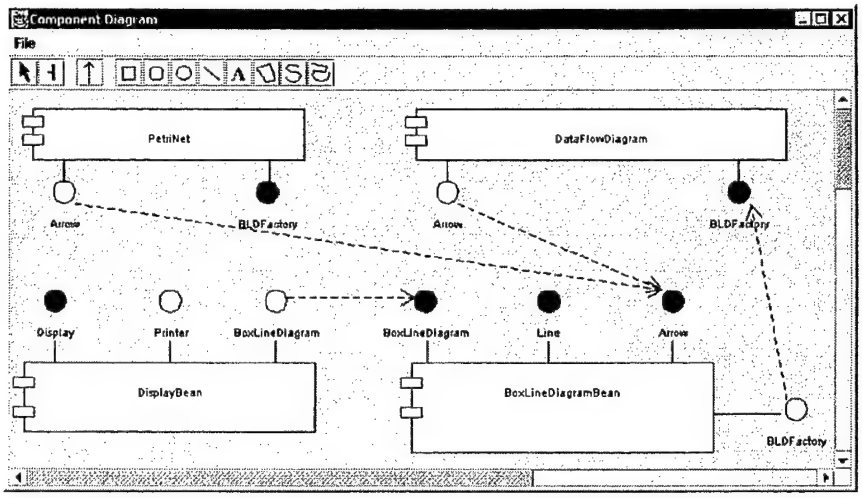


Figure 4: Type-Oriented Component Configuration.

Instance-Oriented Configuration and Component Deployment

The CBDE uses Web Gain Visual Café for instance-oriented configuration. Visual Café is a commercial Java development environment that supports visual connection and adaptation of Java Beans on an instance basis. When the type-oriented configuration described above is completed, the composer can export the components to Visual Café. The environment uses Remote Method Invocation (RMI) to communicate with a Visual Café plug-in, which automatically loads the components into the component library of Visual Café, from where they can be dragged into Visual Café's visual editor.

To make it possible to run component applications, the Java runtime environment is extended by a small library which can interpret component connections and adaptations. To export the configuration information, the environment generates an additional component, the *project component*. It consists of a single class, which encapsulates the mapping of *requires* ports to *provides* ports. When it is executed, it restores the type configuration. When one of the other components is executed and needs one if its required implementations, the extended Java runtime environment will obtain a new instance of the relevant data type from the connected component.

Component Evolution

When a component is marked as selected, the stub can choose between two strategies to provide access to the implementation of the component. In the usual case, it downloads a copy of the implementation and caches it locally, so that method calls can be executed without significant delay. Then, it subscribes with the repository for update notifications. When an updated version of the component is published at the repository, the stub is notified and can update the cached copy.

The other possible strategy is service reuse [7]. Analogous to a client-server application architecture, the stub forwards method calls to the master copy of the component that is located at the repository. Since the component is encapsulated, the difference between the two strategies is transparent to the user of the component. This means that the component can decide at runtime which strategy to use. For example, when the network transfer rate is high enough, the most current data can be retrieved from the remote server. At times when the network is overloaded, the stub can decide to use the locally cached copy of the implementation.

Both these strategies realize reuse by reference. In both cases, a logical connection between the application using a component and the original copy of the component is created in order to prevent the maintenance problems associated with reuse.

4 RELATED WORK

While CBDEs have yet to become a focus of widespread research, there are several previous research efforts that contribute technologies, principles and insights for CBDE design.

An overview of the history and possible future of software engineering environments is given by Harrison et al. [8]. They consider *multi-view software environments* to be one of the most promising recent trends.

Every complex system has many concerns that have to be considered separately. This can only be done by providing different, independent views of the various aspects of a system. Type and instance view in WREN are examples of two views that show different aspects of the same system.

The ArchStudio project [13], which evolved out of the Arcadia project [9] and work on the C2 architectural style [22], defines an event-based architecture for a family of software engineering environments. The architectural style used lends itself to distribution, but it is still a subject of current research to determine whether this is possible on an Internet scale. However, integration of WREN with ArchStudio is planned. While tool integration in WREN is currently implemented on an ad-hoc basis, the principled approach of ArchStudio is clearly preferable.

Koala [24] is a component model for embedded software in consumer electronics. It uses an explicit, visual description of architectures based on the architecture description language Darwin [11]. Like Darwin, it has *provides* and *requires* interfaces and treats interfaces as first-class entities. While Darwin was originally geared towards distributed systems, Koala demonstrates the usefulness of these features in a reuse-oriented component model.

The Application Web [15] is a strategy for sharing information between cooperating organizations that tries to minimize the problems caused by copying over organizational borders. Instead, connections are created to reuse data. Connections make it possible to automate caching, and to access all (not just part of) the context in which the data were originally created. Connections are comparable to the component references discussed in this paper.

The Basic Interoperability Data Model (BIDM) [3], developed by the Reuse Library Interoperability Group (RIG), is a standard for repositories of reusable artifacts that interoperate. The aim is to provide access to all artifacts offered by a network of repositories through any one repository, thus building a decentralized repository There are two preconditions for this: There has to be a standard for meta-information about the artifacts, and a way to uniquely identify artifacts. The proposed data model covers some of the aspects we are suggesting for component self-description; however, the information is not stored in the component itself. Uniform Resource Names (URN) are the proposed solution for the identification problem; since a standard for URNs has not been adopted yet, URLs are used. In this way, the naming scheme is effectively equivalent to the naming conventions for Java packages that we rely on.

Whitehead et al. [25] point out that a well-designed architecture is an essential prerequisite for any component marketplace. They identify criteria for such an architecture, the most important of which are realized in WREN as follows:

- *Multiple component granularities* are given in WREN through the possibility to encapsulate any number of classes into a component.
- *Substitutability of components* is realized through the exclusive use of Java interfaces to specify component dependencies. Every interface can be implemented by any number of components, so that every component is substitutable.
- *Easy distribution of components* from seller to buyer is realized by the integration of development environment and component repository.

Brownsword et al. [4] share our view that new processes for developing component-based systems must be defined. Similar to Morisio et al. [14], they stress that the use of licensed components whose source code cannot be modified influences both requirements and design. Since there is a trade-off between the choice of components to license and the requirements and design of the system, these three issues have to be considered simultaneously.

Alpha Services [7] make applications available through the Internet. Instead of downloading and installing a program, it is accessed through the network when needed. This is a kind of reuse by reference; instead of components, services are reused. Candidates for Alpha Services are functionalities that are hard to develop, infrequently used, and can be modeled as transactions, for example natural language translation or large-scale optimization.

The Software Dock [6] is a system supporting the software deployment lifecycle. It integrates producer-side activities such as releasing and retiring a product with consumer-side activities such as installing, updating and uninstalling. Similar to WREN, a permanent connection is established between consumer and producer side. The Software Dock uses SRM [23] to administer the dependencies among application parts, which are administered by the components themselves in our system. Similar to a CBDE, SRM is geared towards applications made up from independently produced parts.

5 CONCLUSIONS

In this paper we have motivated the need for a new generation of software environments to support the special needs of component-based development. We identified seven important requirements for CBDEs, and we described a prototype environment called WREN that we are building to implement these requirements and to provide a basis for further evaluation and study of the role of environment technology in component-based development.

There are several issues that remain to be resolved. Type-based adaptation does not exist yet in our prototype. Current tools provide mechanisms to adapt component instances, but not components themselves. We expect that the same methods of internal and external adaptation can be used in varied forms. Integration with development environments is another issue. It remains to be seen if tight integration of the CBDE with a commercial development environment is the optimal solution, or if a more specific solution is needed..

Updating of components still requires manual effort. While the environment can automatically retrieve updates, it cannot update components that are being used in an application. Doing so will probably require support for dynamic architecture modification [16]. Another important issue is contract negotiation. A component may be able to dynamically decide about trade-offs between quality of service and price, for example, so that it can negotiate with another component or a human who wants to use this component. Negotiating will require explicit environment support, so that a user can define minimum requirements, policies, and so on.

ACKNOWLEDGMENTS

This effort was sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under grant number CCR-9701973. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U. S. Government.

REFERENCES

1. Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. Making Components Contract Aware. *Computer* 32, 7 (1999), 38-45.
2. Bosch, J. Adapting Object-Oriented Components. In *Object-Oriented Technology.* Springer, Berlin, 1998, 379-383.
3. Browne, S. V., and Moore, J. W. Reuse Library Interoperability and the World Wide Web. *Software Engineering Notes* 22, 3 (1997), 182-189.
4. Brownsword, L., Oberndorf, T., and Sledge, C. A. Developing New Processes for COTS-Based Systems. *IEEE Software* 17, 4 (2000), 48-55.
5. Dashofy, E. M., Medvidovic, N., and Taylor, R. N. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 3-12.
6. Hall, R. S., Heimbigner, D., and Wolf, A. L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proc. 1999 International Conference on Software Engineering.* ACM, New York, 1999, 174-183.
7. Harrison, C. G., and Stern, E. H.: Alpha Services—an Experiment in Developing Enterprise Applications. Accessed Aug. 2000 at http://www.isr.uci.edu/events/twist/twist2000/statements/harrison-stern.doc. 2000.
8. Harrison, W., Ossher, H., and Tarr, P. Software Engineering Tools and Environments: A Roadmap. In *The Future of Software Engineering.* ACM, New York, 2000, 261-277.
9. Kadia, R. Issues Encountered in Building a Flexible Software Development Environment—Lessons from the Arcadia Project. *Software Engineering Notes* 17, 5 (1992), 169-180.
10. Krueger, C. W. Software Reuse. *ACM Computing Surveys* 24, 2 (1992), 131-183.

11. Magee, J., and Kramer, J. Dynamic Structure in Software Architectures. *Software Engineering Notes* 21, 6 (1996), 3-14.

12. Maurer, P. M. Components: What If They Gave a Revolution and Nobody Came? *Computer* 33, 6 (2000), 28-34.

13. Medvidovic, N., Oreizy, P., Taylor, R. N., Khare, R., and Guntersdorfer, M.: An Architecture-Centered Approach to Software Environment Integration. Accessed Aug. 2000 at ftp://www.ics.uci.edu/pub/arch/papers/TR-UCI-ICS-00-11.pdf. 2000.

14. Morisio, M., Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E., and Condon, S. E. Investigating and Improving a COTS-Based Software Development Process. In *Proc. 2000 International Conference on Software Engineering.* ACM, New York, 2000, 32-41.

15. Murer, T., and Van De Vanter, M. L. Replacing Copies with Connections: Managing Software across the Virtual Organization. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'99).* IEEE Computer Society, Los Alamitos, 1999, 22-29.

16. Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (1999), 54-62.

17. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (1972), 1053-1058.

18. Robbins, J. E., and Redmiles, D. F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology* 42, (2000), 79-89.

19. Selic, B., and Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Accessed Aug. 2000 at http://www.rational.com/media/whitepapers/umlrt.pdf. 1998.

20. Shaw, M., and Garlan, D. *Software Architecture.* Prentice Hall, Upper Saddle River, 1996.

21. Szyperski, C. *Component Software.* ACM, New York, 1997.

22. Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Jr., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22, 6 (1996), 390–406.

23. van der Hoek, A., Hall, R. S., Heimbigner, D., and Wolf, A. L. Software Release Management. In *Proc. Sixth European Software Engineering Conference.* Springer, Berlin, 1997, 159-175.

24. van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. *Computer* 33, 3 (2000), 33-85.

25. Whitehead, E. J., Robbins, J. E., Medvidovic, N., and Taylor, R. N. Software Architecture: Foundation of a Software Component Marketplace. In *Proc. First International Workshop on Architectures for Software Systems.* ACM, New York, 1995, 276-282.

26. Zaremski, A. M., and Wing, J. M. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology* 4, 2 (1995), 146-170.

Extending Component Interoperability Standards to Support Architecture-Based Development

Rema Natarajan *David S. Rosenblum*

Technical Report 98-43
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

1 December 1998

Extending Component Interoperability Standards to Support Architecture-Based Development

Rema Natarajan **David S. Rosenblum**
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
1-949-824-6534
{rema,dsr}@ics.uci.edu

ABSTRACT
Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of standard component interoperability models, such as ActiveX, CORBA and JavaBeans. In academia, there has been intensive research in developing a notion of software architecture. Both of these efforts use software components as the basic building blocks, and both address concerns of structure and reuse. With component interoperability models, the focus is on specifying interfaces, binding mechanisms, packaging, inter-component communication protocols, and expectations regarding the runtime environment. With software architecture, the focus is on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components. Our research involves studying how standard component models can be extended to accommodate important issues of architecture, including a notion of architectural style and support for explicit connectors. For our initial effort in this work, we have enhanced the JavaBeans component model to support component composition according to the C2 architectural style. Our approach enables the design and development of applications in the C2 style using off-the-shelf Java components or "beans" that are available to the designer. In this paper, we describe the techniques underlying our approach, and we identify the important issues that surface when attempting this type of extension.

Keywords
Architectural style, C2, component standards, connectors, JavaBeans, software architecture

1 INTRODUCTION

Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of component interoperability models, such as ActiveX [1], CORBA [12], and JavaBeans [14]. These models help developers deal with the complexity of software and facilitate reuse of off-the-shelf components. Component interoperability models also make a positive move toward standardization of components, and the creation of a software component marketplace.

Software architecture research deals with the same issues of software complexity and promoting reuse. Software architecture has been the focus of intense research in academia. Architectures help designers focus on system level requirements and the interconnection of components in a large-scale software system.

Both these approaches use software components as the fundamental building blocks. With component interoperability models, the focus is on specifying interfaces, packaging, binding mechanisms, inter-component communication protocols, and expectations regarding the runtime environment. With software architectures and architectural styles, the focus is on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components [9].

We have begun studying how standard component models can be leveraged to accommodate issues of architectural modeling, including a notion of architectural style and support for explicit connectors. As described in this paper, our approach merges component interoperability models with suitable architectural styles to leverage the full benefit from both technologies, and to develop a comprehensive approach to software development.

We have chosen the JavaBeans component interoperability model as our initial platform for investigation. We made this choice for a variety of reasons:

- The Java language and the JavaBeans component model are becoming increasingly popular and have been widely adopted as de facto standards.
- JavaBeans tools and resources are free or have negligible cost.

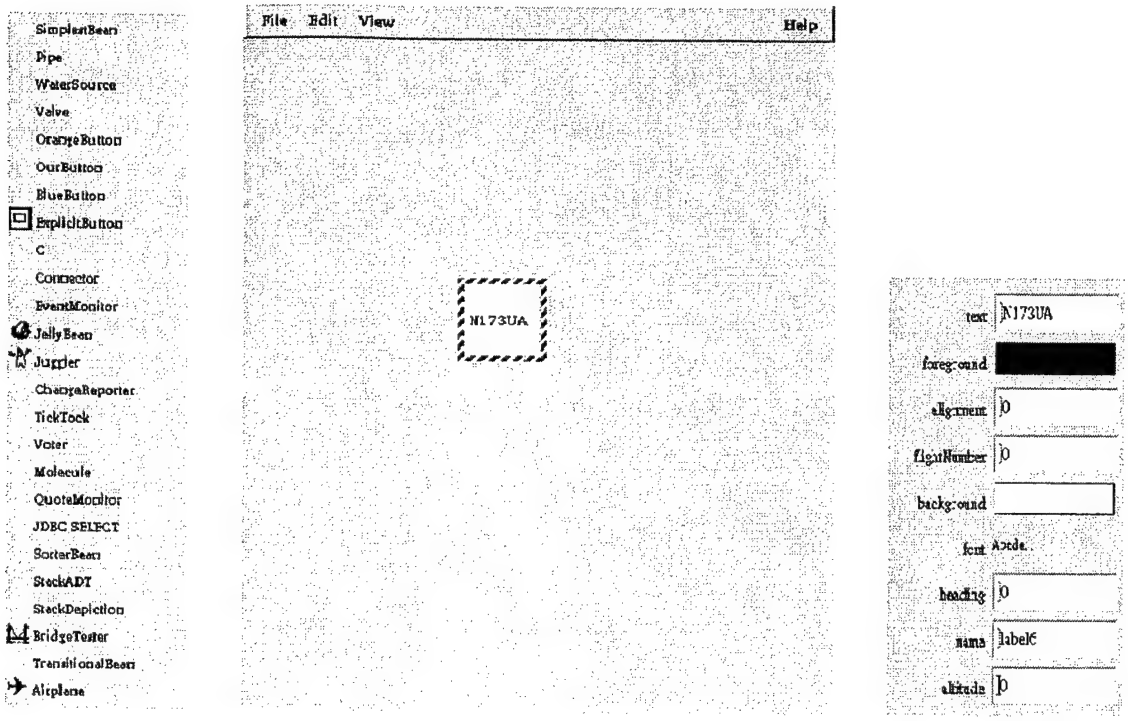


Figure 1. The Sun BDK JavaBeans design environment, with theToolBox of available beans shown on the left, the BeanBox design palette shown in the center with an Airplane bean, and the Property Sheet of the Airplane bean shown on the right.

- The model of composition in JavaBeans is natural and straightforward.
- JavaBeans is a lightweight and flexible framework that lends itself to modification and extension.

In addition, we have chosen the C2 architectural style as our initial architectural technology because it is a novel style that is highly flexible and lends itself naturally to dynamic architectural change. It also supports the property of substrate independence that facilitates reuse and substitutability across architectures with ease of effort.

In this paper, we describe our work in enhancing the JavaBeans component model to support component composition according to the C2 architectural style. Our approach enables the design and development of applications in the C2 architectural style using off-the-shelf Java components or *beans* that are available to the designer. The creation of individual components with their specific interfaces, functionalities and behaviors is a different task from the composition of an architecture of a system that satisfies requirements. The merging of the component interoperability model with the architectural style provides a seamless integration of both activities.

2 THE JAVABEANS COMPONENT MODEL

The JavaBeans component model is a component interoperability model tailored to the Java language. Interoperability is achieved primarily by designing component or *bean* interfaces according to a *component design pattern*.[1] The JavaBeans design pattern defines a naming scheme and interaction protocol to which compliant beans must adhere. The interface constituents governed by this design pattern include properties, methods, and events that together define a bean interface. *Properties* encapsulate key attributes of a bean and can be *simple*, *bound* (meaning they generate events whenever they change values) or *constrained* (meaning their changes can be vetoed by other beans). *Methods* are public operations that form part of the bean interface. Beans communicate with each other through bean *events*; the event handling is based on the Java 1.1 event model.

Figure 1 depicts the JavaBeans design environment that is provided by Sun Microsystems in their Beans Development Kit (BDK) [2]. The environment allows designers to develop beans using the JavaBeans design pattern and to instantiate and test bean compositions. This environment is

[1] The term *design pattern* has been used by many authors to characterize the JavaBeans interface design convention, even though it does not correspond to the usual notion of a design pattern as a frequently-recurring design solution [3].

```
import java.awt.*;
import java.beans.*;

public class Airplane extends Label {
 protected int altitude = 0;
  protected PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

  public int getAltitude() { return altitude; }

  public void setAltitude(int a) {
    int oldAltitude = altitude;
    altitude = a;
    changes.firePropertyChange("Altitude", new Integer(oldAltitude),
                               new Integer(a));
    repaint();
  }

  public void addPropertyChangeListener(PropertyChangeListener l) {
    changes.addPropertyChangeListener(l);
  }

  public void removePropertyChangeListener(PropertyChangeListener l) {
    changes.removePropertyChangeListener(l);
  }
}
```

Figure 2. Partial Java code for the example Airplane bean. The code demonstrates how a bean property is declared plus the methods interested beans can use to listen to bean property change events.

representative of the kinds of visual design environments that can be used to support construction of applications with the JavaBeans component model. As shown in the figure, a JavaBeans design environment includes a palette of available beans (called the ToolBox in the BDK environment), a design tablet on which beans are instantiated and interconnected (called the BeanBox in the BDK environment), and a Property Sheet showing the properties of the bean that is currently selected in the BeanBox.[2]

For instance, consider an Airplane bean that represents the control of an airplane. A partial design for an Airplane bean class is presented in Figure 2. As shown in the figure, the class declares properties that will show up in the property sheet of the BeanBox. In particular, the Airplane declares a property called Altitude, which the BeanBox will find because the class exports the pair of methods getAltitude() and setAltitude() (following the naming scheme of the JavaBeans design pattern). As shown in Figure 1, this property appears in the Property Sheet for the Airplane bean (as do other properties, whose implementation is not shown in Figure 2). While it is possible for changes to the property to be effected by manually coding calls to these exported methods, the BeanBox is designed to allow the designer to customize bean properties via the Property Sheet.

The Altitude property is a bound property because it fires a PropertyChange event whenever its value changes during a call to the method setAltitude(). Other beans register and unregister themselves as listeners for this event by calling the methods addPropertyChangeListener() and removePropertyChangeListener(), respectively; these methods are required by the JavaBeans design pattern when bound properties are to be supported. While it is possible for such event-based interactions between the Airplane bean and other beans to be coded by hand, the BeanBox has been designed to support "wiring up" interacting beans in a graphical manner. In particular, the designer would select the Airplane bean with the pointing device, select a menu item corresponding to the PropertyChange event, and then select a target bean to receive the event; the BeanBox would then take care of generating the necessary method calls to effect the interaction. Note that each bean maintains its own set of listeners and manages event notification by itself.

As can be seen in Figure 2, the JavaBeans design pattern paves the way for building tools that can dynamically "introspect" beans and publish their interfaces, in a manner similar to what the BDK BeanBox does. The tools can provide designers with the capability for customizing bean behavior. Additionally, the JavaBeans design pattern

[2] In this paper we use the term BeanBox as a synonym for a complete JavaBeans design environment.

defines a notion of *bean customizers*, which can be built to allow complex customization of a bean's appearance and behavior, and *property editors*, which define custom editors for a specific property. These two mechanisms aid the design and implementation of generic beans that can be easily customized for different applications. Apart from their individual customizability, beans can be composed in the BeanBox to create running applications, and their runtime behavior can be tested during the design phase itself. This novel capability blurs the distinction between "design-time" and "runtime", since manipulating beans in this manner actually has the effect of creating running instances of bean classes that cooperate according to the designer's intent.

In this manner, the JavaBeans component model concentrates on the interface a Java software building block can or should present. It does not specify how the building blocks can or should be combined to create any kind of application. It specifies how two or more beans can communicate information, without imposing any semantic rules on the information exchanged or on the topology of any bean communication network [14]. The JavaBeans design pattern is designed to make development tools better aware of component capabilities; in particular, the interface pattern has been defined for a modern software developer who will manipulate beans via visual interactions.

3 THE C2 ARCHITECTURAL STYLE

The C2 architectural style is primarily concerned with high-level system composition issues, rather than particular component packaging approaches [9,13]. The building blocks of C2 architectures are components (computational elements) and connectors (interconnection and communication elements). This separation of computation from communication enables the construction of flexible, extensible, and scalable systems that can evolve both at design-time and runtime. The style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing the use of multiple interoperability technologies for its connectors. This flexibility has enabled us to use the event-based interoperability of JavaBeans for our purposes.

Central to the C2 style is the principle of limited visibility or *substrate independence*: components are arranged in a layered fashion in a C2 architecture, and a component is completely unaware of components that reside beneath it in the stack of component layers. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. Components communicate only by exchanging messages through connectors, which greatly simplifies the problem of control integration issues; this property also facilitates low-cost interchangeability of components to construct different members of the same system family. Two components cannot assume that they will execute in the same address space; this eliminates complex dependencies, such as components sharing global variables and simplifies modification of architectures. Conceptually, components run in their own thread(s) of control, allowing components with different threading models to be integrated into a single application. Finally, a conceptual C2 architecture can be instantiated in a number of different ways. Many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation techniques.

C2 components and connectors have a notion of a "top" and a "bottom" interface through which they receive and send messages and communicate with other components in the architecture. This notion of a "top" and a "bottom" is important to ensure substrate independence. Messages that travel up the architecture are called *requests*, and messages that travel down the architecture are called *notifications*. Components execute application logic and communicate with other components in the architecture via requests and notifications. Components do not communicate directly with one another, but instead must communicate through connectors that take care of most of the management of message traffic in the system. Connectors, on the other hand, can be directly connected to each other.

The advantage of explicit connectors is that they encapsulate the logic for message broadcasting, message filtering, and other interaction logic, and they reduce the complexity involved in composing components. This is different from the JavaBeans model where every bean manages its event listeners and event propagation on its own. Additionally, the notion of connectors supports a more generic structure whereby it becomes easier to substitute one component or connector with another, and enables reuse of individual components or connectors across different architectures. It also becomes easier to support dynamic alteration of the architecture. The C2 style constraints (which we have only briefly summarized here) help preserve these C2 properties.

4 A C2-AWARE COMPOSITION ENVIRONMENT

We have begun our investigation of the problem of merging component models with architectural styles by enhancing the BDK BeanBox described in Section 2. In our approach, we create beans using the same JavaBeans design pattern, thus retaining all the advantages of the beans component model. However, we extend the JavaBeans model to incorporate the notion of "components" and "connectors" as defined in the C2 architectural style, and we extend the BeanBox composition functionality to enforce the rules of the C2 style. Thus, we have enhanced the BeanBox and made it "C2-aware".

C2 Style Beans

Our C2-Aware BeanBox provides two mechanisms to

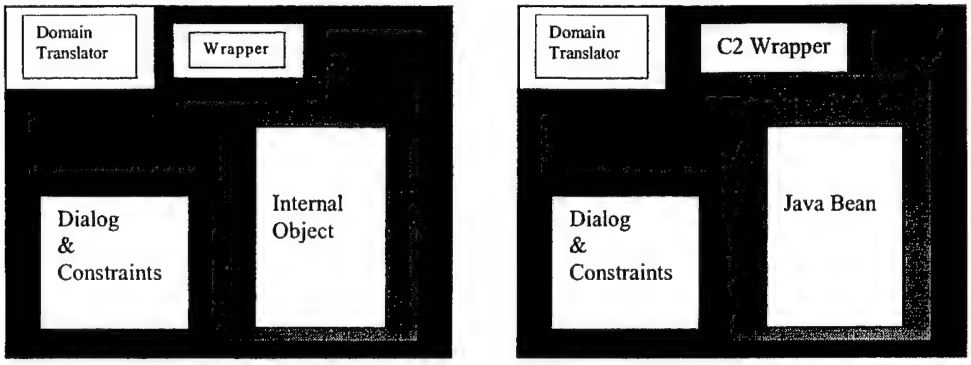


Figure 3. Wrapping of C2 components; the general C2 model of wrapping is shown in the picture on the left, while the picture on the right shows how the general model has been applied for JavaBeans.

instantiate C2 components and C2 connectors as beans. In our first approach, we have provided C2 component and connector "framework beans" that can be subclassed by developers interested in creating beans for specific applications. When these beans are instantiated in the BeanBox, they automatically publish their C2 interface and can thus be hooked up "as is" to compose applications. Beans created in this fashion are fully C2 compatible, and this approach is convenient for developers to create new beans in a way that incorporates the characteristics of a C2 architecture described in Section 3.

In our second approach, we have provided a C2 wrapping mechanism to create C2 components from off-the-shelf beans. Figure 3 presents the model component wrapping we use, which follows the general model of wrapping that has been developed for the C2 style [7,13]. Off-the-shelf beans that have already been developed by other vendors can be instantiated into the BeanBox. Upon this instantiation, a wrapper object is created for the bean to make it C2 compliant. This C2 wrapper is created with the help of interactive dialogs that publish the interface of the bean and guide the designer in mapping the bean's events into C2 requests and notifications. The wrapper then uses this information to build the internal dialog component that is responsible for converting incoming requests and notifications into bean events. The *domain translator* component of the C2 wrapper is used to resolve incompatibilities between communicating components such as mismatches between message names, parameter types and ordering of parameters. The *constraints* component specifies constraints that cannot be violated by the component, provides recovery mechanisms when constraints are violated and exceptions are raised, and provides mechanisms to customize the bean so that constraints are satisfied without raising exception conditions.

Most of the translation required for converting beans into C2 components involves mapping bean events to requests and notifications in the C2 style. The properties that a bean publishes in its property sheet are used "as is" after the bean has been wrapped as a C2 component. Other tools provided for manipulation of beans such as property editors and bean customizers can also be used "as is" in the C2 aware BeanBox.

This second approach has several advantages. Existing beans can be used off-the-shelf simply by plugging them into the BeanBox with the help of the C2 wrapper. The wrapper handles the translation of bean events into C2 messages. Additionally, the dialog and constraints can be used to build in constraints for the component as specified in an architecture specification to ensure that the bean is properly customized for the specific architecture. It is also the facility that lends itself most naturally to providing dynamic testing, analysis and instrumentation mechanisms. We plan to focus on these issues much more in the near future.

The C2-Aware BeanBox

C2 compliant beans, which can be created using any of the two approaches described above, can be instantiated into the C2-Aware BeanBox, as shown in Figure 4. The C2-Aware BeanBox has all the C2 style rules and constraints built into it. It provides a C2 Style Dialog that notifies designers whenever stylistic constraints are violated and thus guides the designer through the composition process. Components and connectors are hooked up using the bean event wiring mechanism, where request events and notification events become the two kinds of events that beans use to interoperate. As required by the C2 style, components cannot be connected to other components, and connectors handle the propagation of events. Thus, unlike in the traditional beans model, beans composed in the C2-Aware BeanBox do not maintain lists of other bean

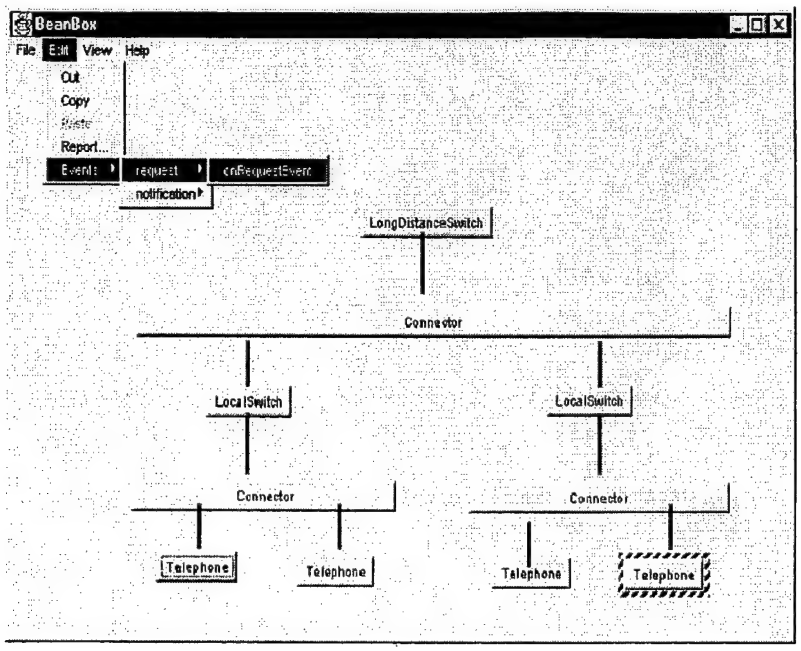


Figure 4. The C2-Aware BeanBox. The figure depicts the composition of a telephone network application as a group of beans interacting in the C2 style.

listeners or handle notification of events to those listeners. Instead, event notification is handled by C2 connector beans. Hence, component bean behavior is better confined to the execution of application logic.

The C2-Aware BeanBox thus allows one to build complex compositions of beans in the C2 style as different instantiations of a given C2 architecture. Introspection mechanisms employed in the C2-Aware BeanBox are used to extract the properties, methods and events that form the public interface of the bean. Conceptually, beans communicate using bean events; these events then become the requests and notifications in the C2 architecture. The designer informs the C2-Aware BeanBox through an appropriate dialog about how events are to be classified as requests and notifications and then manages the communication of beans through these requests and notifications.

5 AN EXAMPLE JAVABEANS-BASED C2 ARCHITECTURE

We have chosen a telephone network system as an example to illustrate our approach; the instantiation of the application in the C2-Aware BeanBox is depicted in Figure 4. Our hypothetical system consists of telephones, local switches and long distance switches. Each of these components is represented as standard beans that publish events (such as ring, hang up, busy) and properties (such as phone numbers and area codes). The properties are bound properties, and thus they fire PropertyChange events. In our C2 architecture for the telephone system, the telephones form the lowest layer of the architecture (i.e., the "interface elements", as is typical of C2-style architectures), with local switches in a layer above the telephones, and the long distance switches at the highest layer in the design.

Upon instantiation of the beans into the C2-Aware BeanBox, each bean gets wrapped in a C2 wrapper. As described in Section 4, the dialog component of the C2 wrapper dynamically introspects the bean and then displays the bean's events in a list and lets the user select the events that should get published as requests and those that should get published as notifications for that bean component. For example, for the telephone component, we would select the "dial" event as a request that needs to travel "up" in the architecture, and the "ring" event as a notification event that needs to travel "down" the architecture. We use standard connectors that are provided as part of the C2 framework to link the components of the telephone network together. The connector propagates request events fired by a component connected to its bottom interface to all components attached to its top interface. Thus a request event for dialing a number by a telephone is propagated through the connector above it to the local switch that handles requests for that area in the architecture. The local switch in turn forwards the request to the long distance switch above it. The long distance switch forwards the

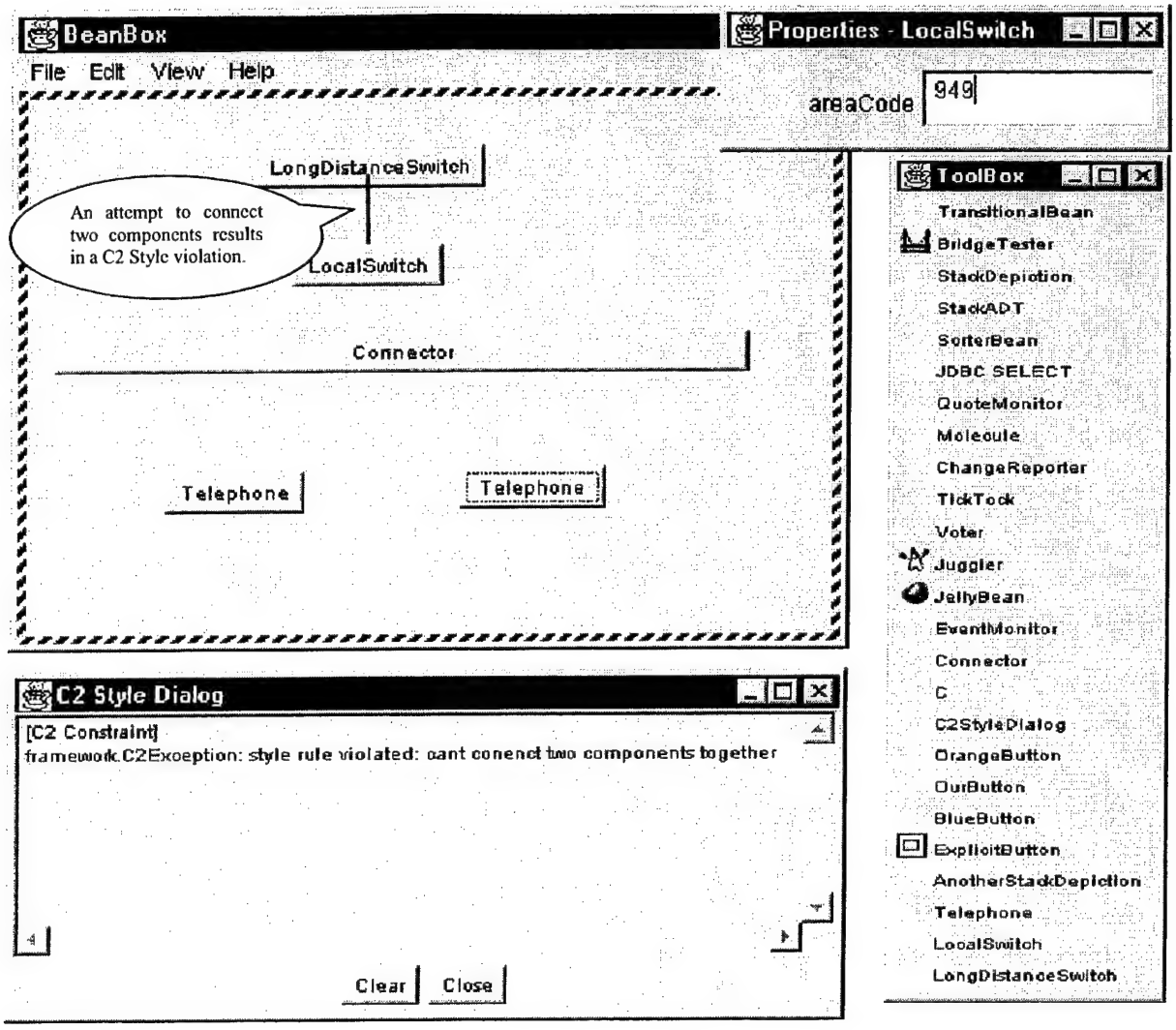


Figure 5. Wiring up the Long Distance Switch and the Local Switch in the C2-Aware BeanBox. This throws a C2 Style Violation exception, which appears in a C2 Style Dialog. The Property Sheet of the Local Switch shows its Area Code property.

message "down" the architecture as notifications to the local switches below it. The local switch with the area code for the dialed number processes the notification by generating a "ring" event as a notification for the telephones below it. The telephones receive the notification, and the telephone with the correct number responds to the notification by processing it. The beans themselves retain their interfaces as before, but the wrapper ensures that beans effectively communicate with the rest of the C2 architecture.

As shown in Figure 5, beans are instantiated and removed from the architecture easily using C2-Aware BeanBox. As the telephone network is built by plugging beans into the architecture, the C2-Aware BeanBox makes automatic checks to ensure that C2 stylistic rules are honored. For example, an attempt to link two telephones directly will raise an exception message in a popup window, helping the designer through the process of composing the system. Figure 5 shows how an attempt to link two components—the Long Distance Switch and the Local Switch—throws an exception that brings up the C2 Style Dialog.

6 DISCUSSION AND RELATED WORK

A lot of interesting issues came up in our effort to create plug-and-play functionality with off-the-shelf beans in our C2-Aware BeanBox. Here we discuss these issues with respect to some of the Component Integration Heuristics for C2 [7].

- If the OTS (off-the-shelf) component does not contain all of the needed functionality, its source code must be altered. While it is interesting to think of situations where other components might be used in conjunction with the OTS component (without altering its source code) to provide the needed functionality, this would be a complex task to attempt, and would depend on the type of functionality that is required.
- If the OTS component does not communicate via messages, a C2 wrapper must be built for it. This facility has already been provided in our C2-Aware BeanBox for all OTS beans that are used as C2 components in a C2 architecture. The wrapper does all the translation necessary to make the OTS bean C2 compliant.
- If the OTS component is implemented in a programming language different from that of other components in the architecture, an IPC (interprocess communication) connector must be employed to enable their communication. As we have solely dealt with the Java language and the BeanBox environment, this issue does not arise in our work.
- If the OTS component must execute in its own thread of control, an inter-thread connector must be employed.
- If the OTS component communicates via messages, but its interface does not match interfaces of components with which it is to communicate, a domain translator must be built for it. We have done some preliminary work in providing domain translation, and we are currently working on improving this support.

Apart from these heuristics described in [7], there are other interesting issues that have arisen:

- If an OTS component provides the functionality required in the C2 architecture, there is still the necessity, in a development and testing environment such as the C2-Aware BeanBox to provide mechanisms to test and validate the architecture instantiation against an architecture specification. Right now the Dialog and Constraints component of the C2 Wrapper provides no support for this. As we discuss in Section 7, we tend to explore this problem in the future.
- The C2-Aware BeanBox facilitates design and composition of systems using any OTS beans that are available. There are exciting possibilities to be explored in strengthening support for design at the architectural level, apart from the work we have already done for the C2 style. For example, we could use the same design environment to create "architecture template beans" for a required C2 architecture, specifying the interfaces of the C2 components and connectors. The template beans could then be used directly, and the same visual environment could be used to populate the templates with OTS beans. This then reduces the work needed for creating the wrapper for OTS beans, and we can use the same architecture templates to create different instantiations of an architecture family. As we discuss in Section 7, this could be done by leveraging the ADL and environment described in [8].

There has been little work to date on supporting architectural modeling in conjunction with standard design technologies. C2, Darwin and UniCon are examples of ADLs that provide a proprietary implementation infrastructure to support an associated ADL. C2 has its class framework as its infrastructure, and this class framework is implemented in multiple programming languages [13]. Darwin is supported by an infrastructure called Regis for distributed programs that are configured using Darwin [4,5]. And UniCon supports implementation generation for a predefined collection of connectors [11]. There has also been recent work in the Darwin project on supporting architectural modeling of CORBA-based systems [6].

In addition to providing ADL-specific infrastructure support, there has been recent work on incorporating substantial support for architectural modeling into the Unified Modeling Language (UML), an emerging standard design notation [10].

7 CONCLUSIONS

Having considered and explored the possibility of combining a popular component interoperability model with a useful software architectural style, we are convinced of the advantages of this approach in the development of component-based software. The philosophy of substrate independence in C2 makes substitution of components and reconfiguration of architectures fairly easy. These modifications to the architecture are done with the least amount of effort because we leverage the strengths of the JavaBeans component model and the ability of the C2-Aware BeanBox to dynamically publish the interface of a beans component and map it to C2-style interactions. The use of a wrapper separates the application logic in the bean component from the translation and dialog with other architectural components. Our C2-Aware BeanBox is a powerful design environment that lets us develop different architectural instantiations with the ease of using a visual environment. It is an example of a tool where the distinction between the design environment and the runtime environment of systems has become blurred.

A key advantage of our approach is that our architectural infrastructure is now complete, to the extent that the full range of developmental activities is supported from the

design, implementation and adaptation of individual components, to the design, implementation and integration of architectures that are compositions of these individual elements. Another advantage is that all these activities are now integrated into a single environment, and this leads the way to a seamless, comprehensive development philosophy that facilitates easy shifting of focus from one activity to another. Sophisticated architectural development tools built along these lines will tie in neatly with component-based software development.

In the future, we plan to further investigate the issues raised and opportunities opened up by this approach. The ability to test the runtime behavior of bean components in a design environment is extremely useful for test different architectural configurations. As we discussed in Section 4, a natural place to provide instrumentation support for testing is in the dialog and constraints portion of the C2 wrapper shown in Figure 3. As also discussed in that section, we would like to begin supporting checking of component semantic constraints in a manner that respects emerging approaches to architectural modeling and emerging standards for component interoperability. An excellent starting point for this work would be to leverage two recent additions to the C2 arsenal, the ADL C2 SADEL and its associated environment DRADEL, which support modeling and evolution of architectures according to a rich model of heterogeneous subtyping of component interfaces [8]. Finally, we need to find ways of adapting our visual approach for architectural construction to distributed architectures. A current limitation of the JavaBeans model is that it does not support composition of distributed beans that must communicate via remote procedure call (e.g., using Java Remote Method Invocation). With the C2-Aware BeanBox, we can naturally incorporate such mechanisms for distributed interaction within C2 connectors, yet we must provide additional support for specifying deployment of beans across distributed hardware.

Our experience with JavaBeans and C2, we believe, are helping to us expand and develop our understanding of the synergy between component models and software architectures.

ACKNOWLEDGEMENTS

Discussions with Dick Taylor, Peyman Oreizy, Elisabetta Di Nitto and Alfonso Fuggetta helped us improve many of the ideas presented in this paper. This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602- 97-2-0021; by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] D. Chappell, *Understanding ActiveX and OLE.* Redmond, WA: Microsoft Press, 1996.

[2] A. DeSoto, "Using the Beans Development Kit 1.0: A Tutorial", JavaSoft, Sun Microsystems, Inc., Mountain View, CA November 1997.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[4] J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment for Distributed Programs", *IEE/IOP/BCS Distributed Systems Engineering*, vol. 1, no. 5, pp. 304–312, 1994.

[5] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

[6] J. Magee, A. Tseng, and J. Kramer, "Composing Distributed Objects in CORBA", *Proc. Third International Symposium on Autonomous Decentralized Systems*, Berlin, Germany, pp. 257–263, 1997.

[7] N. Medvidovic, P. Oreizy, and R.N. Taylor, "Reuse of Off-the-Shelf Components in C2-Style Architectures", *Proc. 19th International Conference on Software Engineering*, Boston, MA, pp. 692–700, 1997.

[8] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", in submission August 1998.

[9] P. Oreizy, N. Medvidovic, R.N. Taylor, and D.S. Rosenblum, "Software Architecture and Component Technologies: Bridging the Gap", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA January 1998.

[10] J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method", Department of Information and Computer Science, University of California, Irvine, Irvine, CA, Technical Report 97-35, November 1997.

[11] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[12] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[13] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[14] L. Vanhelsuwe, *Mastering JavaBeans*: SYBEX Inc, 1997.

An Architecture-Based Approach to Self-Adaptive Software

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf

SELF-ADAPTIVE SOFTWARE REQUIRES HIGH DEPENDABILITY, ROBUSTNESS, ADAPTABILITY, AND AVAILABILITY. THIS ARTICLE DESCRIBES AN INFRASTRUCTURE SUPPORTING TWO SIMULTANEOUS PROCESSES IN SELF-ADAPTIVE SOFTWARE: SYSTEM EVOLUTION, THE CONSISTENT APPLICATION OF CHANGE OVER TIME, AND SYSTEM ADAPTATION, THE CYCLE OF DETECTING CHANGING CIRCUMSTANCES AND PLANNING AND DEPLOYING RESPONSIVE MODIFICATIONS.

CONSIDER THE FOLLOWING SCENARIO. A fleet of unmanned air vehicles undertakes a mission to disable an enemy airfield. Pre-mission intelligence indicates that the airfield is not defended, and mission planning proceeds accordingly. While the UAVs are en route to the target, new intelligence indicates that a mobile surface-to-air missile launcher now guards the airfield. The UAVs autonomously replan their mission, dividing into two groups—a SAM-suppression unit and an airfield-suppression unit—and proceed to accomplish their objectives. During the flight, specialized algorithms for detecting and recognizing SAM launchers automatically upload and are integrated into the SAM-suppression unit's software.

In this scenario, new software components are dynamically inserted into fielded, heterogeneous systems without requiring system restart, or indeed, any downtime. Mission replanning relies on analyses that include feedback from current performance. Furthermore, such replanning can take place autonomously, can involve multiple, distributed, cooperating planners, and where major changes are demanded and require human approval or guidance, can cooperate with mission analysts. Throughout, system integrity requires the assurance of consistency, correctness, and coordination of changes.

Other applications for fleets of UAVs might include environment and land-use monitoring, freeway-traffic management, fire fighting, airborne cellular-telephone relay stations, and damage surveys in times of natural disaster. How wasteful to construct afresh a specific software platform for each new UAV application! Far better if software architects can simply adapt the platform to the application at hand, and better yet, if the platform itself adapts on demand even while serving some other purpose. For example, an airborne sensor platform designed for environmental and land-use monitoring could prove useful for damage surveys following an earthquake or hurricane, provided someone could change the software quickly enough and with sufficient assurance that the new system would perform as intended.

Software engineering aims for the systematic, principled design and deployment of applications that fulfill software's original promise—applications that retain full plasticity throughout their lifecycle and that are as easy to modify in the field as they are on the drawing board. Software engineers have pursued many techniques for achieving this goal: specification languages, high-level programming languages, and object-oriented analysis and design, to name just a few. However, while each contributes to the goal, the sum total still falls short.

Self-adaptive software will provide the key. Many disciplines will contribute to its progress, but wholesale advances require a sys-

1094-7167/99/$10.00 © 1999 IEEE

tems perspective based on a broadly inclusive adaptation methodology that spans a wide range of adaptive behaviors. Central to our view is the dominant role of software architecture in planning, coordinating, monitoring, evaluating, and implementing seamless adaptation. This article examines the fundamental role of software architecture in self-adaptive systems and outlines technologies we have considered for supporting the methodology.

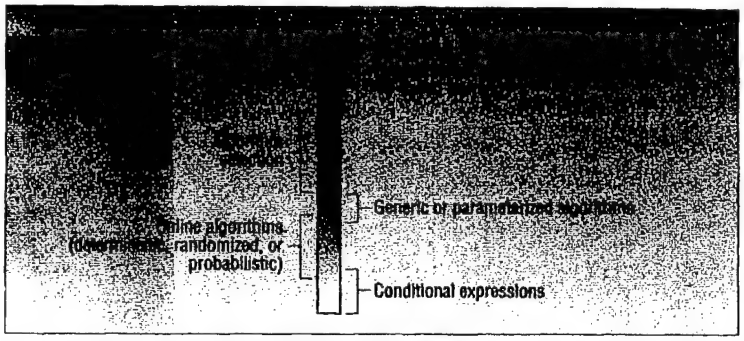


Figure 1. A spectrum of self-adaptability. Generally, approaches near the bottom select among predetermined alternatives, support localized change, and lack separation of concerns. Approaches near the top support unprecedented changes and provide a clearer separation of software-adaptation concerns.

What is self-adaptive software?

Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.

Application developers must answer several questions when developing a self-adaptive software system:

- Under what conditions does the system undergo adaptation? A system might, for example, modify itself to improve system response time, recover from a subsystem failure, or incorporate additional behavior during runtime.
- Should the system be open-adaptive or closed-adaptive? A system is open-adaptive if new application behaviors and adaptation plans can be introduced during runtime. A systems is closed-adaptive if it is self-contained and not able to support the addition of new behaviors.
- What type of autonomy must be supported? A wide range of autonomy might be needed, from fully automatic, self-contained adaptation to human-in-the-loop.
- Under what circumstances is adaptation cost-effective? The benefits gained from a change must outweigh the costs associated with making the change. Costs include the performance and memory overhead of monitoring system behavior, determining if a change would improve the system, and paying the associated costs of updating the system configuration.
- How often is adaptation considered? A wide range of policies can be used, from opportunistic, continuous adaptation to lazy, as-needed adaptation.
- What kind of information must be collected to make adaptation decisions? How accurate and current must the information be? A wide range of strategies can be used, from continuous, precise, recent observations to sampled, approximate, historical observations.

What conditions? A fleet of UAVs might undergo adaptation under a variety of conditions. Mission replanning is a prime example because automated or human mission planners redirect the fleet in response to the changing battlefield. A mechanical failure of a UAV's generator might force the UAV to rely solely on battery power for its electronics, communications, and sensors. This in turn would require substantial adaptation to ensure sufficient electrical power for the mission's duration. A change in force composition (such as the loss of a fleet member to equipment failure) or the detection of an unanticipated threat might force rapid and substantial adaptation.

Open- or closed-adapted? A closed-adaptive UAV adapts in isolation, uninfluenced by the adaptations and behaviors of other fleet members. It has only a limited number of adaptive behaviors onboard, and no new behaviors can be introduced at runtime. Such a UAV might be capable of a limited number of evasive maneuvers in response to threats, for example, and its repertoire of evasions cannot be modified or expanded in flight. Conversely, an open-adaptive UAV accepts behaviors introduced from the outside, so an evasive maneuver known to one fleet member can be communicated to others while in flight.

Type of autonomy? Each UAV can be autonomous to a greater or lesser degree. For example, a UAV coping with an inflight subsystem failure might require that a human-in-the-loop direct, or at least approve, an adaptation. A sophisticated UAV with more onboard computing power might be highly autonomous, interacting with humans infrequently, if at all, over the course of its mission.

Frequencies? Adaptation is not without its cost, and even a useful or desirable adaptation might require more resources than the UAV can afford. For example, the UAV might be forced to permanently discard applications or system support for the sake of additional memory to accommodate an adaptation, or the adaptation might cut off future avenues of change. Implementing the adaptation might require processor cycles better used for other, more pressing concerns, or the adaptation, though desirable, might degrade the UAV's performance in other respects.

Cost-effectiveness? Adaptation frequency also matters. A UAV might be opportunistic, considering and implementing adaptations whenever it has spare processor cycles or additional communications bandwidth available. It might also adapt continuously, allocating an ongoing fixed percentage of its computing and communication resources to the adaptation process. Alternatively, adaptation might only be on demand as warranted by the UAV's condition and environmental state.

Information type and accuracy? Finally, the UAV might collect information from numerous sources on which to base its adaptation decisions. Information sources include

- real-time readings from internal sensors for monitoring subsystem health and status (such as battery voltage or fuel levels),
- telemetry from external sensors such as radar and magnetometers,
- sampled observations such as processor

load or radio signal strength over minutes, or historical data such as the movements of threat forces over hours.

Figure 1 illustrates the broad spectrum of self-adaptability. At one extreme, conditional expressions are a form of self-adaptation; the program evaluates an expression and alters its behavior based on the outcome. Although simplistic, conditional expressions are a common mechanism for implementing adaptive behavior. For example, a just-in-time compiler might invoke aggressive code-optimization techniques if a function is called frequently.

Online algorithms operate under the assumption that future events (inputs) are uncertain. Hence, they will occasionally perform an expensive operation to more efficiently respond to future operations.[1] Online algorithms are adaptive in that they leverage knowledge about the problem and the input domain to improve efficiency. A memory-cache-paging algorithm, for example, leverages the spatial and temporal locality of memory references in determining which cached page to evict when making room for a new page.

Generic and parameterized algorithms provide behaviors that are parameterized, usually through type instantiation or external inputs. Generic or polymorphic algorithms adapt by conforming to different data types. The C++ Standard Template Library, for example, provides generic iterator classes used to traverse a variety of data structures.

Algorithm selection uses properties of the operating environment to choose the most effective algorithm among a fixed set of available algorithms. Thus, a system that uses algorithm selection adapts to changes in its operating environment by switching among a set of algorithms. The Self dynamic optimizing compiler, for example, uses program-profiling data collected during program execution to select different code-optimization algorithms.[2]

At the other extreme, evolutionary programming and machine-learning techniques are adaptive in that they use properties of the operating environment and knowledge gained from previous attempts to generate new algorithms.[3]

Generally, approaches near the spectrum's bottom intertwine concerns regarding software adaptation and application-specific behavior. For example, a conditional expression combines the adaptation's specification with the application's specification. Consequently, understanding, analyzing, and modifying the two independently is arduous.

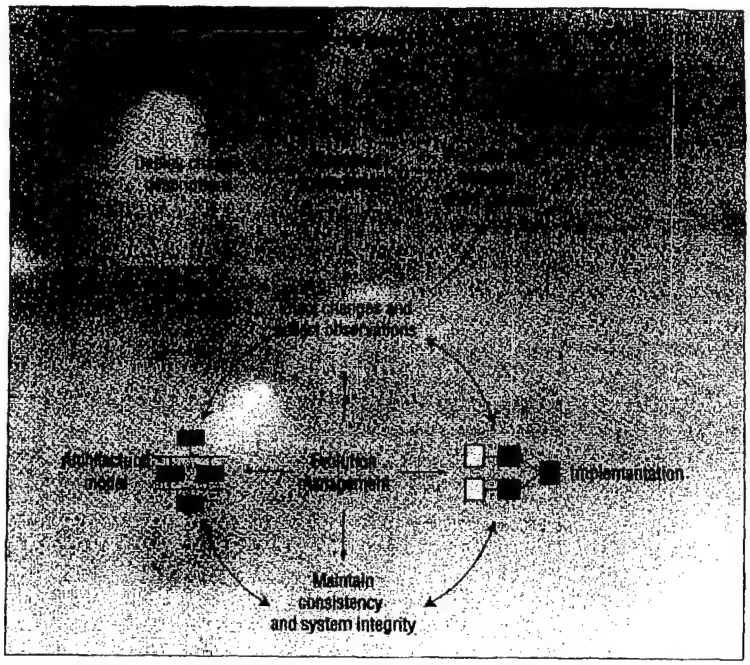


Figure 2. High-level processes in a comprehensive, general-purpose approach to self-adaptive software systems.

Approaches near the top more clearly separate software-adaptation concerns and application-specific functionality. For example, algorithm generation separates the adaptation's specification from the produced algorithm. Separating the concerns of software adaptation from software function facilitates their independent analysis and evolution.

Software adaptation in-the-large

While technical advances in narrow areas of adaptation technology provide some benefit, the greatest benefit will accrue by developing a comprehensive *adaptation methodology* that spans adaptation-in-the-small to adaptation-in-the-large, and then develops the technology that supports the entire range of adaptations. Figure 2 illustrates just such a methodology that we are investigating.

The upper half of the diagram, labeled "adaptation management," describes the lifecycle of adaptive software systems. The lifecycle can have humans in the loop or be fully autonomous. "Evaluate and monitor observations" refers to all forms of evaluating and observing an application's execution, including, at a minimum, performance monitoring, safety inspections, and constraint verification. "Plan changes" refers to the task of accepting the evaluations, defining an appropriate adaptation, and constructing a blueprint for executing that adaptation. "Deploy change descriptions" is the coordinated conveyance of change descriptions, components, and possibly new observers or evaluators to the implementation platform in the field. Conversely, deployment might also extract data, and possibly components, from the running application and convey them to some other point for analysis and optimization.

Adaptation management and consistency maintenance play key roles in this approach. Although mechanisms for runtime software change are available in operating systems (for example, dynamic-link libraries in Unix and Microsoft Windows), component object models, and programming languages, these facilities all share a major shortcoming: they do not ensure the consistency, correctness, or other desired properties of runtime change. Change management is a critical aspect of runtime-system evolution that identifies what must be changed; provides the context for reasoning about, specifying, and implementing change; and controls change to preserve system integrity. Without change management, the risks engendered by runtime modifications might outweigh those associated with shutting down and restarting a system.

Software adaptation is a complex process and is further complicated by change drivers ranging from purposeful adjustments in

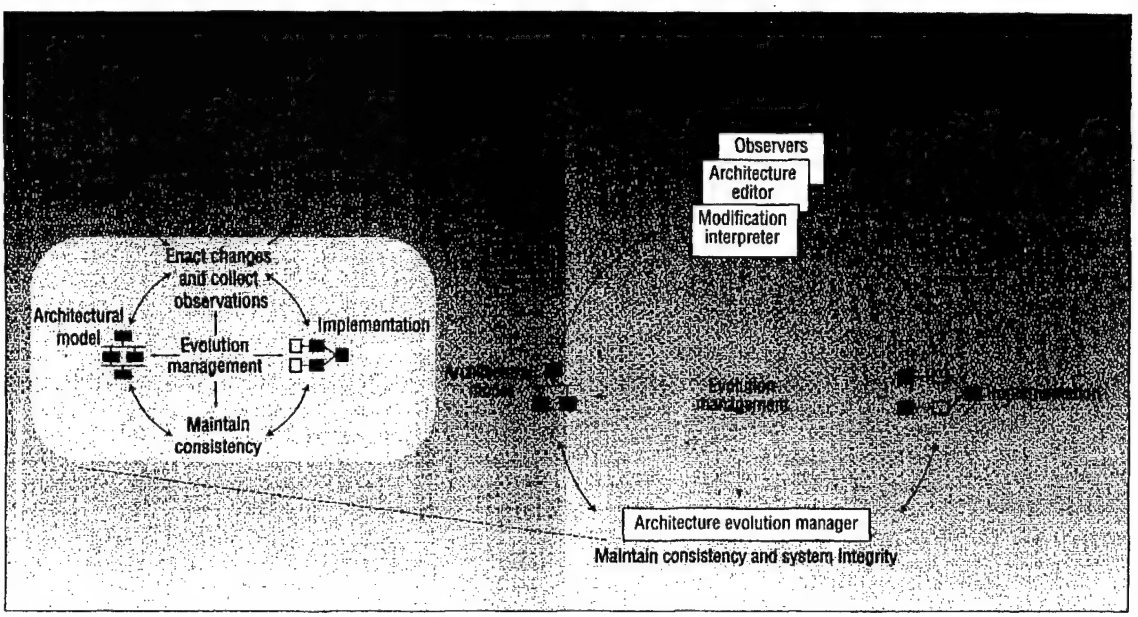


Figure 3. A high-level architecture diagram for the ArchStudio tool suite.

fielded systems to unanticipated perturbations in the operational environment. The changes themselves encompass everything from a simple replacement of an isolated component to wholesale reconfigurations that are pervasive and physically distributed. Our approach addresses these demanding and unprecedented requirements by managing adaptation using a flexible infrastructure to support a full range of adaptation processes. The infrastructure relies on

- software agents that automate tasks within the process,
- explicit representations of software components, their interdependencies, and their environmental assumptions,
- explicit representations of the environments in the field where software is deployed, and
- wide-area messaging and event services that connect adaptation managers to adaptive systems to permit coordinated and coherent adaptation in physically distributed, logically decentralized environments.

The lower half of Figure 2, labeled "evolution management," focuses on the mechanisms employed to change the application software. Our approach is architecture-based: changes are formulated in, and reasoned over, an explicit architectural model residing on the implementation platform. Changes to the architectural model are reflected in modifications to the application's implementation, while ensuring that the model and the implementation are consistent with one another. Monitoring and evaluation services observe the application and its operating environment and feed information back to the diagram's upper half.

Software architectures view systems as networks of concurrent components bound together by connectors.[4] An architectural perspective shifts focus away from source code to coarse-grained components and their interconnections. Designers can abstract away obscuring details and concentrate on the big picture: the system structure, the interactions among components, the assignment of components to processing elements, and runtime change. Components are responsible for implementing application behavior and maintaining state information. Connectors are transport and routing services for messages or objects. Components do not know or care how their inputs and outputs are delivered or transmitted or even what their sources or destinations might be. On the other hand, connectors know exactly who is talking to whom and how—but are ignorant of the computations of the components they serve. Strictly separating computation from communication lets a system's computation and communication relationships evolve independently of one another, including rearranging and replacing the components and connectors of an application while the application executes—a necessary, but insufficient, mechanism for self-adaptive software.

Evolution management

It is not enough that we can rearrange and replace portions of an application while it is executing. Self-adaptive systems present a unique set of challenges with respect to safety, reliability, and correctness. For example, an ill-considered change—such as the accidental removal of a critical navigation component—can compromise a UAV's safety, reliability, and correctness properties. Consequently, facilities for guiding and verifying modifications are an integral part of our architecture-centric approach. Figure 3 details our approach to evolution management, the process by which change is applied and controlled.[5] A variety of tools and adaptation mechanisms evolve an application by inspecting and changing its architectural model. Changes can include the addition, removal, or replacement of components and connectors, modifications to the configuration or parameters of components and connectors, and alterations in the component/connector network's topology. As we show next, our approach maintains system consistency and integrity by examining each change and vetoing any changes that render the system inconsistent or unsafe.

Dynamic software architectures. Supporting a broad class of adaptive changes at the architectural level requires that we not only change components on the fly but also their interconnections. However, simultaneously changing components, connectors, and top-

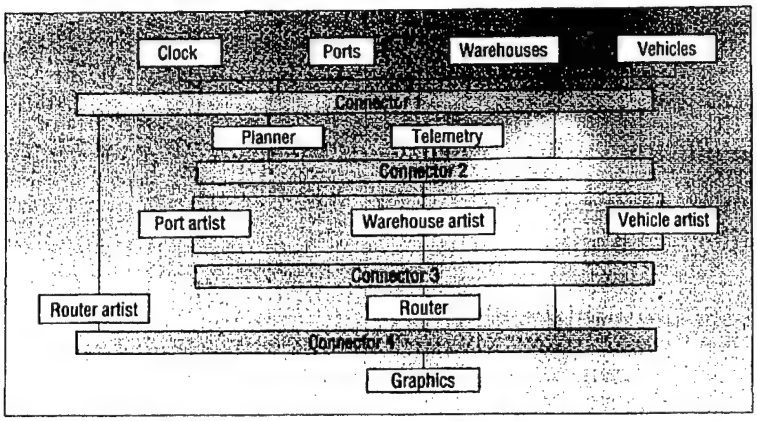


Figure 4. A C2-style architecture for a simple cargo-routing logistics system. Ports, vehicles, and warehouses are components that store application state. The telemetry component tracks en route cargo shipments. The port artist, vehicle artist, warehouse artist, and router artist components graphically depict the state of their respective counterparts. The planner component uses simple heuristics to suggest cargo routes, and the router component handles routing requests initiated by the end user. The graphics component renders the drawing notifications sent from the artists on the end-user's display.

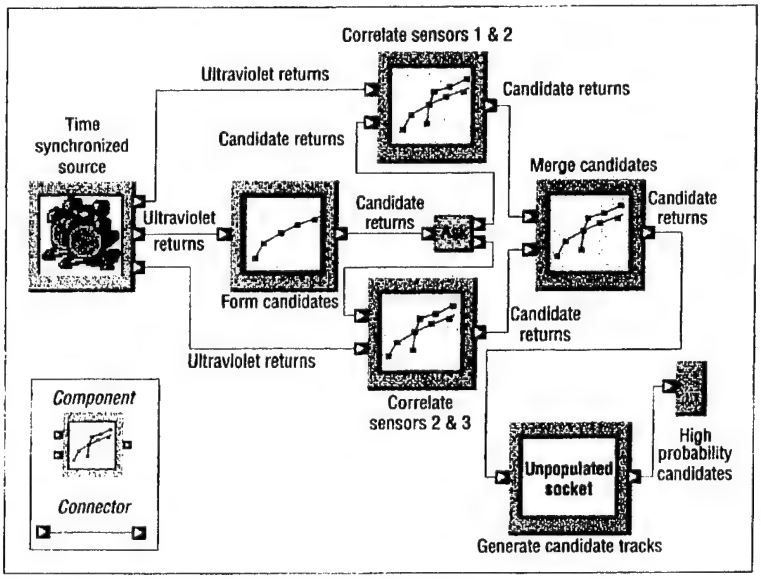


Figure 5. A portion of a Weaves architecture for a stereo-tracking system.

ology in a reliable manner requires distinctive mechanisms and architectural formalisms. Many systems are dynamic to some limited degree but few embrace dynamic change as a fundamental consideration.

There are two distinct approaches to dynamism at the architectural level: C2[6] and Weaves.[7] They have many features in common:

- both distinguish between components and connectors,
- neither places restrictions on the granularity of the components or their implementation language,
- both require that all communication between components occur by exchanging asynchronous messages (C2) or objects (Weaves), and
- components can encapsulate functionality of arbitrary complexity and exploit multiple threads of control.

However, C2 and Weaves take different approaches to system composition. C2 composes systems as a hierarchy of concurrent components bound together by connectors—message-routing devices—such that a component within the hierarchy can only be aware of components "above" it and is completely unaware of components residing at the same level or "beneath" it. Figure 4 shows an example C2-style architecture for a simple cargo-routing logistics system. A component explicitly utilizes the services of components above it by sending a request message. Communication with components below occurs implicitly; whenever a component changes its internal state, it announces the change by emitting a notification message, which describes the state change, to the connector below it. Connectors broadcast notification messages to every component and connector connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to a single component's state change. For example, the "Telemetry" component in Figure 4 is only aware of the "Clock," "Ports," "Vehicles," and "Warehouses" components. Furthermore, the C2-style components cannot assume that they will execute in the same address space as other components or share a common thread of control.

In contrast, Weaves is a dynamic, object-flow-centric architecture designed for applications characterized by continuous or intermittent voluminous data flows and real-time deadlines. Components in Weaves consume objects as inputs and produce objects as outputs ("object" is intended in the sense of C++, Smalltalk, or Java). Figure 5 depicts an example Weaves architecture for a portion of a stereo tracker. Weaves embraces a set of architectural principles known as the laws of blind communication:

- no component in a network knows the sources of its input objects or the destinations of its output objects;
- no network component knows the semantics of the connectors that delivered its input objects or transmitted its output objects; and
- no network component knows the loss of a connection.

These laws ensure that no component knows its location in the network, that every component is independent of the semantics of the connectors to which it is attached, and that any Weaves architecture can be edited, rewired, expanded, or contracted on the fly. Furthermore, Weaves permits connectors to be composed of other connectors and components, allowing connectors to be specially

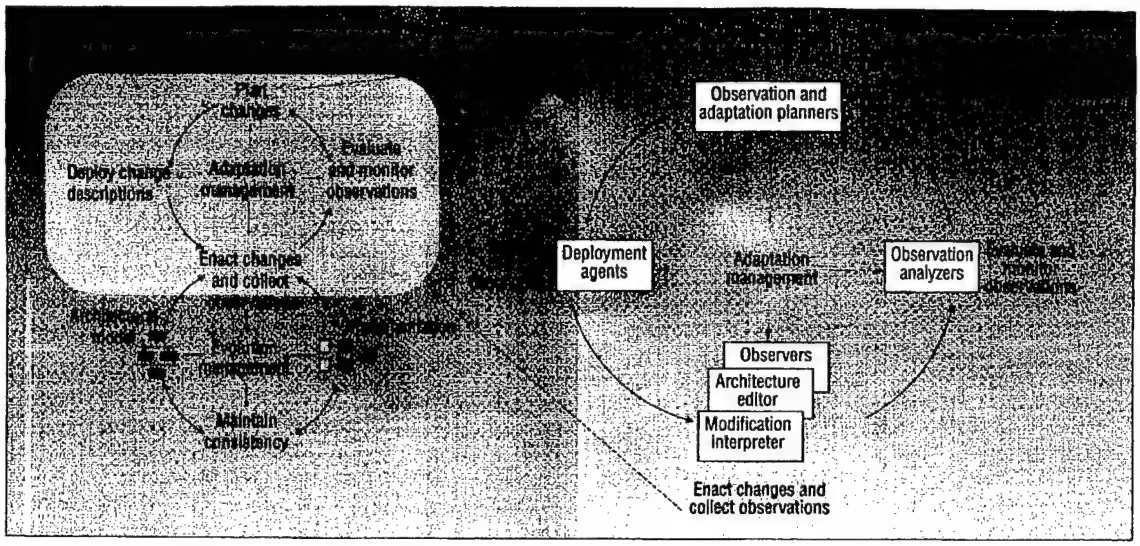


Figure 6. A high-level architecture diagram for adaptation evolution.

adapted to the characteristics of their operating environment with corresponding performance gains.

Several characteristics of C2 and Weaves facilitate runtime change. Because components communicate asynchronously, C2 and Weaves avoid several subtle complexities inherent in supporting runtime change in applications that utilize synchronous communication. While this restriction occasionally makes it more difficult to implement particular component interactions, because a component must continue to respond to service requests from other components while awaiting responses it has made of others, our experience demonstrates that its benefits for runtime change outweigh its costs.

The independence between hierarchical layers in a C2-style architecture further reduces component dependencies: a C2 component is unaware of components below itself, so it is oblivious to runtime changes that involve these components.. Conversely, a component can only be affected by runtime changes involving components strictly above itself. Because C2 components cannot assume that they will execute in the same address space as other components, complex component dependencies resulting from the use of pointer variables and global variables are avoided. Similarly, because components do not share a common thread of control, control dependencies are avoided. Taken together, C2's style rules ensure that each component is almost completely ignorant of the placement, function, and implementation of its fellow components.

Consequently, at runtime, C2 can add, delete, or rearrange components with remarkable ease and alacrity. In contrast, while Weaves supports like forms of component manipulation, it emphasizes the dynamic distribution, modification, and rearrangement of connectors. This lets developers optimize intercomponent communication while a Weaves architecture is executing, including wholesale movement of a subarchitecture from one host to another along with dramatic changes in the semantics and implementations of its connectors.

In short, C2 has been optimized for flexible components, while Weaves focuses on high-performance, flexible connectors. One research issue we face is blending these two approaches to dynamic architectures into a single, cohesive whole. One possible approach is to treat Weaves as an implementation substrate for C2 and "compile" C2-style architectures into lower-level, but more efficient, Weaves architectures.

Maintaining consistency and system integrity. Ongoing adaptation continuously threatens system safety, reliability, and correctness. Therefore, facilities for guiding and checking modifications are an integral part of our adaptation infrastructure. As an application adapts and evolves, we face the problem of preserving an accurate and consistent model of the application architecture and its constituent parts—the components and the connectors. We must also maintain a strict correspondence between the architectural model and the executing implementation. To deal with these problems, we deploy, as an integral part of the application, an architectural model that describes the interconnections among components and connectors and their mappings to implementation artifacts. The mapping permits changes, given in terms of the architectural model, to effect corresponding changes in the actual implementation.

To guard against untoward change, we propose an *architecture evolution manager (AEM)* that mediates all change operations directed toward the architectural model. A change is expressed either as a single basic operation or as a *change transaction* composed of two or more basic operations. All changes are atomic; that is, they either complete without error or leave the application untouched. A change transaction includes operations for forcing components and connectors into safe or halt states; adding, removing, and replacing components and connectors; and changing the architectural topology.

The AEM maintains the consistency between the architectural model and the implementation as changes are applied, reifies changes in the architectural model to the implementation, and prevents changes from violating architectural constraints. For example, it can enforce the generic constraint that all components must be connected to at least one connector but not more than two. The AEM is also tailored by application- and domain-dependent change policies that dictate the forms of acceptable change. Within the UAV domain, the AEM can require that the UAV system contain at least one navigation component. The AEM, which maintains the mapping between the architectural model and the implementation, uses this mapping to carry out modifications by mapping model

components and connectors into implementation artifacts and translating change operations into implementation actions.

Enacting changes. There are many possible sources of architectural change, including the application itself, external tools, and replanning agents. Software architects can use a visual, interactive, *architecture editor* to construct architectures and describe modifications. A variety of analysis tools can accompany the editor—for example, a design wizard that critiques an architecture as a designer constructs it, or application- and domain-dependent design wizards that, by exploiting specialized knowledge, can prevent semantic errors or ensure a minimum level of performance or safety. The *modification interpreter* acts as a second, companion tool to interpret change scripts written in a change-description language to primitive actions supported by the AEM.

Adaptation management

A self-adaptive system observes its own behavior and analyzes these observations to determine appropriate adaptations. A companion to the process of evolution management is the process of adaptation management, illustrated in Figure 6. Adaptation management monitors and evaluates the application and its operating environment, plans adaptations, and deploys change descriptions to the running application.

Viable self-adaptive systems require long-term continuity in the face of dynamic change—in other words, both a standard locale for the information and tasks required to carry out the function of adaptation and a focal point for coordinating physically distributed, logically decentralized adaptation tasks. For example, complex interdependencies might exist among changes such that the incorporation of one change could require the inclusion of several others for the change to work correctly in its environment. A standard locale helps ensure that such information is at hand. Additionally, an adaptation might require coordination among multiple sites when the application is physically distributed and adaptation requires changes at several sites simultaneously.

Additionally, managing self-adaptive software requires a variety of agents, such as *observers* for evaluating the behavior of the self-adaptive application and monitoring its operating environment, *planners* that utilize the observations to plan adaptive responses, and *deployers* to enact the responses within the application.

Hosting the numerous agents and supporting the various activities of adaptation management that result requires infrastructure support in its own right in the form of registries. Registries at each application site contain resource descriptions, configurations, and other declarative information relevant to the site and the adaptive application. Registries elsewhere might be dedicated to overseeing and coordinating the activities of the individual application site registries. Each registry provides a standard interface by which disparate agents and interests can query and manipulate the contents of the registry, which acts as a blackboard for exchanging information. Interregistry communication takes many forms, ranging from directed updates to wide-area messaging and event notification. One promising starting point is the Software Dock, an infrastructure element for the distributed configuration and deployment of software systems, now under development at the University of Colorado, Boulder.[8,9]

MANAGING SELF-ADAPTIVE SOFTWARE REQUIRES A VARIETY OF AGENTS, SUCH AS OBSERVERS, PLANNERS, AND DEPLOYERS. HOSTING THE NUMEROUS AGENTS REQUIRES INFRASTRUCTURE SUPPORT IN ITS OWN RIGHT.

Collecting observations. Self-adaptive software requires large numbers and varieties of observations and measurements, ranging from event-generation within the application implementation to animations suitable for human observers. Furthermore, adjusting the number, extent, and detail of the observations and measurements must be possible as the application executes and evolves so as to reduce measurement overhead and avoid wasting communication bandwidth on unnecessary observations.

At a minimum, we require embedded assertions (inline observers) within the application itself for notification of exceptional events such as resource shortages or the violation of low-level constraints. Additional required capabilities include dynamic control and alteration of the scope of the assertions, insertion and removal of assertions while the application is executing, language-independent assertions, and architecture-sensitive assertions. One potential candidate for this facility is APP, a tool that supports the automated checking of logical assertions expressed in first-order predicate logic.[10]

Detecting and noting single events is not enough because the occurrence of a pattern of events distributed in both time and place will trigger many adaptive strategies. One approach is to model application behavior abstractly in terms of patterns of events. In this way, the architect's expectations are expressed as an *expectation agent*.[11] The expectation agent responds to the occurrence of event patterns, including generating a higher-level abstract event for the benefit of other (expectation) agents. The expectation agent is a formal specification that, depending upon its complexity, can be translated into an observer embedded within the application or implemented as an agent that eavesdrops on the activity of the local registry. In addition, we must monitor events that occur outside of the application—such as the quality or availability of a network connection—as well as adaptation events that arise as a consequence of dynamic architectural change.

We must also make provision for observation by human observers in cooperation with automated agents. One appealing technology is Joist, which exploits standard Web-based technologies—HTTP and HTML—to provide a powerful and efficient infrastructure for remote observation of distributed applications.[12] Joist embeds a small Web server in the application's runtime environment, which then monitors the application and gathers information. This information is identified through a special URL namespace, and it is presented in HTML pages that the Joist server generates and communicates to any standard Web browser via HTTP.

New techniques must emerge for reducing the monitoring overhead. Weaves employs statistical monitoring techniques that lets observers trade accuracy in favor of reduced overhead. Using this approach, we can reduce the invasive effect of instrumentation on a running application to below the noise threshold while still obtaining useful

information. Furthermore, the instrumentation can stay permanently embedded within the application so that an observer can selectively measure only behaviors of interest without damaging the application. Application developers can use this technique in a variety of ways, including performance analysis and real-time animation of the behavior of running systems.

Evaluating and monitoring. Adaptive demands can arise from inconsistencies or suboptimal behavior within the system. In particular, inconsistencies can occur when some architectural element (ranging from a single component or connector to a subsystem, or the entire architecture) behaves in a manner inconsistent with the behavior required of it or when an element's assumptions about its operating environment become invalid. Maintaining consistency in these situations requires monitoring and evaluating representative behaviors of the running system and comparing them to an explicit formulation of behavioral requirements or environmental assumptions.

Successful consistency management requires a hybrid approach that combines both static and dynamic analysis. One promising form of static analysis exploits attributed graph grammars. Recall that dynamic architectures can be characterized as graphs of components and connectors. Attributed graph grammars can represent the set of all an application's possible configurations where architectural changes are regarded as graph-rewrite operations. Analysis tools can determine if an invariant is preserved by all possible architectures or can return an example graph (architectural configuration) that violates the invariant.

Static analysis might be insufficient, in which case runtime checks are employed to detect inconsistencies. Observers inspect both the application and the environment in which the application operates and evaluate their observations for consistency with relevant annotations obtained from the registries. Observers are generated and launched automatically based on the constraints and properties extracted from annotations pertaining to the element under observation. Observers post observed inconsistencies, aggregated observations, and analyses to the registry.

Planning changes. Planning is also a vital aspect of self-adaptive software. Self-adaptation requires two distinct forms of planning: observation planning and adaptation planning. Observation planning determines which observations are necessary for deciding when and where adaptations are required. The observation planner takes into account environmental assumptions, expected behaviors, the availability of observers and observations, and their costs. We can view this task as a classic planning problem where the goals are information needs, the operators are the observers, the preconditions are required event types, the postconditions are observer-generated event types, and the operators have observation and notification costs. This type of planning is well within the range of today's planning technology.

Adaptation planning determines exactly which adaptations to make and when. The adaptation planner must take into account the purpose of components, their environmental assumptions, and known properties of the environment. We are interested in applications (such as UAVs) where adaptations must be planned in minutes, not hours. One possible approach relies on the use of predefined solution frameworks that, by limiting the range and variation of possible adaptations, drastically reduce the computation required for planning.

PLANNING IS A VITAL ASPECT OF SELF-ADAPTIVE SOFTWARE. SELF-ADAPTATION REQUIRES TWO DISTINCT FORMS OF PLANNING: OBSERVATION PLANNING AND ADAPTATION PLANNING.

A solution framework is a partially instantiated hierarchical solution architecture consisting of connectors, sockets (placeholders for components), and an equivalence class of candidate components for each socket (where components can themselves be solution frameworks). Given such a framework, an initial solution architecture might be found by selecting components for each socket from the set of eligible components. Using this as a starting point, the system can undergo incremental adaptation by choosing alternatives for problematic components whose environmental assumptions no longer hold in the observed environment.

A more general approach explicitly represents the preconditions and postconditions of each component that could affect, or be affected by, other components; represents each socket in terms of one or more required postconditions; and treats framework instantiation as a planning problem. This approach requires a partial domain model and additional computation but no longer requires that candidate components form an equivalence class. We have already demonstrated this approach in another domain, the automatic generation of simulation scripts for tank training.[13]

Deploying change descriptions. Change agents propagate and move out among sites to carry out their tasks. Imagine a scenario in which a coordinated change is required at two separate sites. Agents responsible for each portion of the coordinated change dispatch from a third site (which oversees the other two), taking with them the change descriptions to be installed. Included in the change descriptions are any new required components or connectors and their affiliated annotations. These agents, once situated at the registries of their respective sites, will interact with the local AEM, which translates the change transactions contained within the change descriptions into specific modifications of the system's implementation.

ALTHOUGH EACH INDIVIDUAL aspect of our approach has been the focus of much research, integrating these aspects into a comprehensive self-adaptive software methodology is unprecedented. In the near future, we hope to complete an initial integration of our dynamic architecture technology, event-based monitoring and evaluation technology, and software deployment technology in support of self-adaptive software.

Acknowledgments

We thank David M. Hilbert and Jason E. Robbins for discussions that contributed to this work. Dennis Heimbinger and Alexander L. Wolf are sponsored by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency (DARPA) under contracts F30602-94-C-0253 and F30602-98-2-0163. Peyman Oreizy, Richard N. Taylor, Nenad Medvidovic, and David S. Rosenblum are sponsored by DARPA and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement

F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant F49620-98-1-0061; and by the National Science Foundation under grant CCR-9701973. Alex Quilici was partially supported by DARPA contract N66001-96-C-8502. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, the Air Force Office of Scientific Research, or the US Government.

References

1. S. Irani and A.R. Karlin, *On Online Computation: Approximation Algorithms for NP-Hard Problems*, Dorit Hochbaum, ed., PWS Publishing Company, Boston, 1996.
2. U. Holzle, *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*, PhD dissertation, Stanford Univ., Stanford, Calif., 1994.
3. W.M. Spears et al., "An Overview of Evolutionary Computation," *Proc. European Conf. Machine Learning*, Springer-Verlag, New York, 1993, pp. 442–459.
4. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *Software Eng. Notes*, Vol. 17, No. 4, 1992, pp. 40–52.
5. P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution, *Proc. Int'l Conf. Software Eng. (ICSE '98)*, 1998, pp. 117–186.
6. R.N. Taylor et al., "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Trans. Software Eng.*, Vol. 22, No. 6, 1996, pp. 390–406.
7. M.M. Gorlick and R.R. Razouk, "Using Weaves for Software Construction and Analysis," *Proc. Int'l Conf. Software Eng. (ICSE '91)*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 23–34.
8. R.S. Hall et al., "An Architecture for Post-Development Configuration Management in a Wide-Area Network, *Proc. 17th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, 1997, pp. 269–278.
9. R. Hall, D. Heimbigner, and A.L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," *Proc. Int'l Conf. Software Eng., (ICSE '99)*, IEEE CS Press, 1999.
10. D.S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Eng.*, Vol. 21, No. 1, 1995, pp. 19–31.
11. D.M. Hilbert and D.F. Redmiles, "An Approach to Large-Scale Collection of Application Usage Data over the Internet," *Proc. Int'l Conf. Software Eng. (ICSE '98)*, IEEE CS Press, 1998, pp. 136–145.
12. M.M. Gorlick, "Distributed Debugging and Monitoring on $5 a Day," *Proc. California Software Symp.*, Univ. of California, Irvine, Calif., 1997, pp. 31–39.
13. D. Pautler, S. Woods, and A. Quilici, "Exploiting Domain-Specific Knowledge to Refine Simulation Specifications," *Proc. 12th Conf. Automated Software Eng.*, IEEE CS Press, 1997.

Peyman Oreizy is a PhD candidate at the University of California, Irvine. His research interests include software evolution, customization, and architectures. He received his BS and MS in computer science from UCI. He is a member of the IEEE Computer Society and the ACM. Contact him at UC Irvine, Information and Computer Science Bldg., Rm. 444, Irvine, CA 92697-3435; peymano@ics.uci.edu; www.ics.uci.edu/~peymano/.

Michael M. Gorlick is a research scientist at the Aerospace Corporation. His research interests include software architectures, large-scale system and software engineering, and wearable computers. He holds an MSc in computer science from the University of British Columbia, Canada. Contact him at the Aerospace Corp., Mail Station M1-102, PO Box 92957, Los Angeles, CA 90009; gorlick@aero.org.

Richard N. Taylor is a professor with the Department of Information and Computer Science, UCI, and is also the director of the Irvine Research Unit in Software (IRUS), an alliance between California industry and the university. His research interests are centered on software architectures, hypermedia and Web protocols, and workflow and process technologies. He received his PhD in computer science from the University of Colorado, Boulder. He is an ACM Fellow. Contact him at the Dept. of Information and Computer Science, UCI, Irvine, CA 92697-3425; taylor@ics.uci.edu; www.ics.uci.edu/~taylor.

Dennis M. Heimbigner is a research associate professor at the University of Colorado, Boulder. His research interests are in configuration management, paradigms for the engineering of distributed software, distributed computing models, software workflow, and federated databases. He received a BS in mathematics from the California Institute of Technology, and an MS and PhD in computer science from USC. He is a member of the IEEE and ACM, and is a principal investigator in the DARPA EDCS program. Contact him at the Dept. of Computer Science, Campus Box 430, Univ. of Colorado, Boulder, CO 80309-0430; dennis@cs.colorado.edu; www.cs.colorado.edu/~dennis.

Gregory Johnson is a member of the technical staff of Concept Shopping Inc. His interests include software-understanding tools and algorithm visualization. He received his doctorate in computer science from the University of Wisconsin-Madison. He is a member of the IEEE and the ACM. Contact him at 777 Silver Spur Rd., Ste. 229, Rolling Hills Estates, CA 90274; gregfjohnson@earthlink.net.

Nenad Medvidovic is an assistant professor in the Computer Science Department at the University of Southern California. He received his PhD from the Department of Information and Computer Science at UCI. He also received an MS in information and computer science from UCI, and a BS in computer science from Arizona State University. His research interests include software engineering, architectures, evolution, and reuse. Contact him at the Computer Science Dept., Henry Salvatori Computer Center, Rm. 338, Univ. of Southern California, Los Angeles, CA 90089-0781; neno@usc.edu; http://sunset.usc.edu/~neno.

Alex Quilici is an associate professor of electrical engineering at the University of Hawaii, Manoa. His research interests lie in applying AI techniques to software engineering, in particular in the areas of automated program understanding and the automated synthesis of component-based systems. He received his PhD in computer science from UCLA. He is a member of AAAI, the IEEE Computer Society, and the Cognitive Science Society. Contact him at the Univ. of Hawaii, Manoa, Dept. of Electrical Eng., 2540 Dole St., Holmes Bldg., Rm. 483, Honolulu, HI 96822; alex@wiliki.eng.hawaii.edu; www-ee.eng.hawaii.edu/~alex.

David S. Rosenblum is an associate professor in the Department of Information and Computer Science at UCI. His current research is centered on problems in the design and validation of large-scale distributed component-based software systems. He received a PhD from Stanford University. He is a senior member of the IEEE and a member of the ACM. Contact him at the Dept. of Information and Computer Science, UCI, Irvine, CA 92697-3425; dsr@ics.uci.edu; www.ics.uci.edu/~dsr.

Alexander L. Wolf is an associate professor in the Department of Computer Science at the University of Colorado at Boulder. His research interests are directed toward the discovery of principles and development of technologies for supporting the engineering of large, complex software systems. He received a BA in geology and computer science from Queens College, City University of New York, and his MS and PhD in computer science from the University of Massachusetts, Amherst. He is a member of the ACM and the IEEE Computer Society, and is Vice Chair of the ACM Special Interest Group in Software Engineering. Contact him at the Dept. of Computer Science, Campus Box 430, Univ. of Colorado, Boulder, CO 80309-0430; alw@cs.colorado.edu; www.cs.colorado.edu/users/alw.

from the Proceedings of the 21st International Conference on Software Engineering (ICSE 99), Los Angeles, CA, May 16–22, 1999

A Language and Environment for Architecture-Based Software Development and Evolution

Nenad Medvidovic
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-0781
+1-213-740-5579
neno@usc.edu

David S. Rosenblum
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1-949-824-6534
dsr@ics.uci.edu

Richard N. Taylor
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1-949-824-6429
taylor@ics.uci.edu

ABSTRACT

Software architectures have the potential to substantially improve the development and evolution of large, complex, multi-lingual, multi-platform, long-running systems. However, in order to achieve this potential, specific techniques for architecture-based modeling, analysis, and evolution must be provided. Furthermore, one cannot fully benefit from such techniques unless support for mapping an architecture to an implementation also exists. This paper motivates and presents one such approach, which is an outgrowth of our experience with systems developed and evolved according to the C2 architectural style. We describe an architecture description language (ADL) specifically designed to support architecture-based evolution and discuss the kinds of evolution the language supports. We then describe a component-based environment that enables modeling, analysis, and evolution of architectures expressed in the ADL, as well as mapping of architectural models to an implementation infrastructure. The architecture of the environment itself can be evolved easily to support multiple ADLs, kinds of analyses, architectural styles, and implementation platforms. Our approach is fully reflexive: the environment can be used to describe, analyze, evolve, and (partially) implement itself, using the very ADL it supports. An existing architecture is used throughout the paper to provide illustrations and examples.

Keywords

Software architecture, architecture description language, software environment, evolution, implementation mapping.

1 INTRODUCTION

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are commonly estimated to be as high as 60% of overall development costs [9]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus reducing opportunities for reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

An explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key property is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability: few specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [23]. Each of them may evolve. Our work to date has addressed the evolution of connectors and topologies [19, 22, 29, 37]. This paper discusses an approach for evolving software components, which has resulted from the recognition that an existing software module can evolve in a controlled manner via subtyping, as in object-oriented languages (OOPLs), for example. The novel aspects of this approach are:

- a taxonomy that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces;
- a flexible type theory for software architectures that is domain-, style-, and ADL-independent. By adopting a richer notion of typing, this theory is applicable to a broad class of design and reuse circumstances; and
- an approach to establishing type conformance between interoperating components in an architecture, which is better suited than other existing techniques to support the "large-scale development with off-the-shelf reuse" philosophy on which architecture research is largely based.

To support our approach to architecture-based evolution, we have designed a simple ADL that embodies the principles of the type theory. The ADL is accompanied by an environment

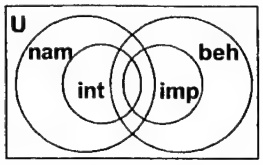


Fig. 1. A framework for understanding OO subtyping relationships as regions in a space, U, of type systems.

for architecture specification, analysis, and evolution. The environment also provides tool support for partial generation of an application from its architecture, which, in turn, facilitates reuse of off-the-shelf (OTS) components. The environment itself is component-based; its architecture was designed to be easily evolvable to support multiple ADLs, types of analysis, architectural styles, and implementation platforms. Our approach is fully reflexive: the environment can be used to describe, analyze, evolve, and (partially) implement itself, using the very ADL it supports.

The remainder of this paper is organized as follows. Section 2 summarizes the general principles of the architectural type theory, as well as the types of evolution and analysis the approach supports [21]. Section 3 introduces the ADL with the help of an example. Section presents and discusses the environment for modeling, analyzing, evolving, and implementing architectures described in the ADL. Section 5 discusses related work. A summary of our results to date, conclusions, and future work round out the paper.

2 THE TYPE THEORY

Motivation

Explicit treatment of types enables *subtyping*, the evolution of a given type to satisfy new requirements, and *type checking*, the determination of whether instances of one type may be legally used in places where another type is expected. A useful overview of PL subtyping relationships is given by Palsberg and Schwartzbach [31]. They describe a consensus in the OO typing community regarding a range of OO typing relationships. *Arbitrary subclassing* allows any class to be declared a subtype of another. *Name compatibility* demands that there exist a shared set of method names available in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass preserve the interface of the superclass, while possibly extending it. *Behavioral conformance* [2, 5, 15, 40] allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. Finally, *strictly monotone subclassing* additionally demands that the subtype preserve the particular implementations used by the supertype.

Protocol conformance goes beyond the behavior of individual methods to specify constraints on the order in which methods may be invoked. Explicitly modeling protocols has been shown to have practical benefits [1, 28, 39, 40]. However, component invariants and method preconditions and postconditions can be used to describe all state-based protocol constraints and transitions. Thus, our notion of behavioral conformance implies protocol conformance, and we do not address them separately.

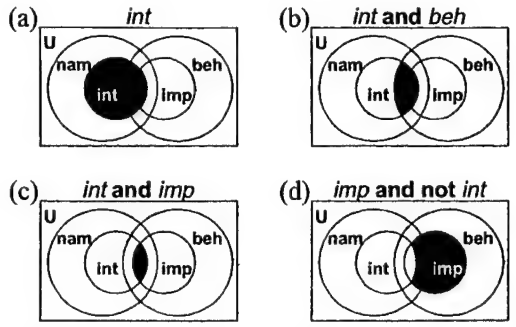


Fig. 2. Examples of component subtyping relationships we have encountered in practice.
(a) *interface conformance* — useful in interchanging components that communicate, e.g., via asynchronous message passing, without affecting dependent components in an architecture;
(b) *behavioral conformance* — guarantees correctness during component substitution;
(c) *strictly monotone subclassing* — enables extension of existing component's behavior while preserving correctness relative to the rest of the architecture;
(d) *implementation conformance with different interfaces* — allows a component to be fitted into an alternate domain of discourse (e.g., by using software adaptors [39]).

We have developed a framework for understanding these subtyping relationships as regions in a space of type systems, shown in Fig. 1. Each subtyping relationship described in [31] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the *int* and *beh* regions (Fig. 2b).

This framework was motivated by our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [18, 22]. Several examples are shown in Fig. 2. Note that we referred to the first three examples (Fig. 2a-c) using the terminology from the Palsberg-Schwartzbach taxonomy. However, while in OOPLs three different subtyping mechanisms would be provided by three separate languages, in architectures such heterogeneous subtyping mechanisms may need to be supported by the same ADL and may all be applied to components in a single architecture. A new type may also be created by subtyping from several existing types using different subtyping mechanisms. Finally, the example in Fig. 2d does not have a corresponding OOPL mechanism, further motivating the need for a flexible type theory for software architectures.

At the same time, by giving a software architect more latitude in choosing the direction in which to evolve a component, we allow some potentially undesirable side effects. For example, by preserving a component's interface, but not its behavior, the component and its resulting subtype may not be interchangeable in a given architecture. However, it is up to the architect to decide whether to preserve architectural type correctness, in a manner similar to America [2], Liskov and Wing [15], Leavens et al. [5], and others (depicted in Fig. 2b), or simply to enlarge the palette of design elements in a controlled manner, in order to use them in the future.

Component Subtyping

Every component specification at the architectural level is an *architectural type*. We distinguish architectural types from *basic types* (e.g., integers, strings, arrays, records, etc.). Unlike OOPLs, in which objects communicate by passing around other objects, in software architectures components are distinguished from the data they exchange during communication. In other words, a "component" in the sense in which we use it here is never passed from one component in an architecture to another.

A component has a *name*, a set of *interface elements*, an associated *behavior*, and (possibly) an *implementation*. Each interface element has a direction indicator (*pro*vided or *req*uired), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type.

A component's behavior consists of an *invariant* and a set of *operations*. The invariant is used to specify properties that must be true of all component states. Each operation has preconditions, postconditions, and (possibly) a result. Like interface elements, operations can be *pro*vided or *req*uired. Only provided operations will have an implementation in a given component. The pre- and postconditions of required operations express their *expected* semantics.

Since we separate the interface from the behavior, we define a mapping function from interface elements to operations. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation's result type.[1] This property directly enables a single operation to export multiple interfaces. An interface element and its corresponding operation comprise a *component service*.

Subtyping Relationships

Given this definition of a component, it is possible to specify component subtyping relationships. We summarize those relationships here. Their formal specification is given in [21].

Name conformance requires that a subtype share its supertype's interface element and parameter names. We have encountered name conformance in practice only as part of the stronger *interface* conformance relationship. Component C_2 is an interface subtype of C_1 if and only if it specifies *at least* the provided and *at most* the required interface elements from C_1, preserving contravariance of parameters and covariance of result.

Behavior conformance requires that the invariant of the supertype be ensured by that of the subtype. Furthermore, each provided operation of the supertype must have a corresponding provided operation in the subtype, where the subtype's operation has the same or weaker preconditions, same or stronger postconditions, and preserves result covariance. This relationship is reversed for required interface elements.

1. We do not explicitly model the semantics of basic types. However, we do allow the specification of subtyping relationships among them.

The subtyping relationship that results from the combination of behavior and interface conformance represents a point in the region depicted in Fig. 2b. This relationship is similar to other notions of behavioral subtyping [2, 5, 15] since it guarantees substitutability between a supertype and a subtype component in an architecture. Our approach is unique in that it establishes this relationship for *required* as well as provided services.

Finally, *implementation* conformance may be established with a simple syntactic check if the operations of the subtype have identical implementations as the corresponding operations of the supertype. On the other hand, establishing semantic equivalence between syntactically different implementations is undecidable in general. Techniques for making this problem tractable are outside the scope of our research.

Type Checking a Software Architecture

There is no single accepted set of guidelines for composing architectural elements into an architecture. Instead, topology depends on the ADL in which the architecture is modeled, the application domain, and/or the rules of the chosen architectural style. In specifying architectures, we adhere to the rules of the C2 style [37]: we model connectors explicitly, limit a component's attachments to single connectors on its top and bottom sides, and allow a connector to be attached to multiple components and connectors. None of these specific choices is required by our type theory. It is indeed possible to provide a definition of architecture that reflects other compositional guidelines. However, these kinds of decisions are necessary in order to formally specify and check type conformance criteria.

It is now possible to specify type checking predicates. A service provided by one component will satisfy a service required by another if and only if their interfaces match, the precondition of the required operation implies the precondition of the provided operation, and the reverse of this relationship holds for their postconditions. As already discussed, components need not be able to fully interoperate in an architecture. The two extreme points on the spectrum of type conformance are:

- *minimal type conformance*, where at least one service required by each component is provided by some other component along its communication links; and
- *full type conformance*, where every service required by every component is provided by some component along its communication links.

Depending on the requirements of a given project (reliability, safety, budget, deadlines, etc.), type conformance corresponding to different points along the spectrum may be adequate. Architectural type correctness is thus expressible in terms of a *degree* of conformance, rather than the "all or nothing" approach found in PLs.

Type Conformance and OTS Reuse

Establishing whether two components can interoperate includes matching the specification of what is expected by a required operation of one component against what another component's provided operation supplies. Behavior of an operation is modeled in terms of its interface parameters and component state variables. A component may thus need to refer to state variables that belong to another component in order to specify a *required* operation's expected behavior. However, doing so would be a violation of the "provider"

component's abstraction. It would also violate some basic principles of component-based development:

- the designer may not know in advance which components, if any, will contain a matching specification for a required operation and, thus, what the appropriate (types of) state variables are. This is particularly the case when using behavior matching to aid component discovery/retrieval;
- large-scale, component-based development treats an OTS component as a black box, thereby intentionally hiding the details of its internal state. Having to explicitly refer to those details would require them to be exposed.

Existing approaches to behavior modeling and conformance checking have not addressed this problem. The problem does not apply to component subtyping: the designer must know all of existing component's details in order to effectively evolve it. Thus, those approaches that focus on behavioral subtyping (e.g., America [2], Liskov and Wing [15], and Leavens et al. [5]) do not encounter this problem. Zaremski and Wing [40] do address component retrieval and interoperability. However, their approach makes the very assumption that the designer will have access to a "provider" component's state (via a shared Larch trait [11]). Meyer [25] makes a similar assumption: all users of a component (class) have full access to the specification of its behavior. Fischer and colleagues [6, 35] model components at the level of a single procedure. In order to be able to properly specify pre- and postconditions, they include all the necessary variables as procedure parameters. Thus, for example, to implement a stack *push* procedure, the stack itself is passed as a parameter to the procedure.

The solution we propose to this problem is based on two requirements arising from a more realistic assessment of component-based development:

- we do not have access to a "provider" component's internal state (unlike Zaremski and Wing's approach), and
- we cannot change the way many software components, especially in the OO world, are modeled (unlike Fischer et al.).

We model a required operation as if we have access to a "provider" component's state. However, since we do not know the actual "provider" state variables or their types, we introduce a generic type, STATE_VARIABLE; variables of this type are essentially placeholders in logical predicates. When matching, say, a required and provided precondition, we attempt to unify (instantiate) each variable of the STATE_VARIABLE type in the required precondition with a corresponding state variable in the provided precondition. If the unification is possible and the implication holds, then the two preconditions conform.

3 THE LANGUAGE

The basic syntactic constructs needed in an ADL to support our type theory were introduced in [18]. We elaborate on those constructs here. This section introduces the basic concepts of C2SADEL (*S*oftware *A*rchitecture *D*escription and *E*volution *L*anguage), a language designed specifically to support architecture-based evolution. C2SADEL supports component evolution via heterogeneous subtyping and facilitates architectural descriptions that allow establishment of type-theoretic notions of architectural soundness. Before presenting the language constructs, we briefly describe an example architecture used for illustration purposes in the remainder of the paper.

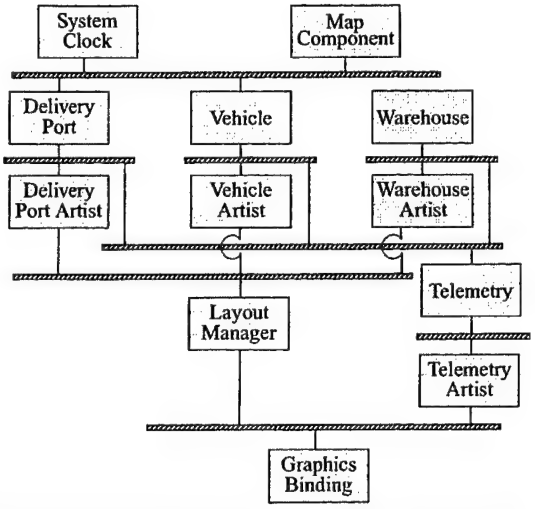


Fig. 3. Architecture of the cargo routing system in the C2 style.

Example Architecture

The architecture we use is a variant of the logistics system for routing incoming cargo to a set of warehouses, first introduced in [29] and shown in Fig. 3. The *DeliveryPort*, *Vehicle,* and *Warehouse* component types are objects that keep track of the state of a port, a transportation vehicle, and a warehouse, respectively. Each of them may be instantiated multiple times in a system. The *DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist* components are responsible for graphically depicting the state of their respective objects to the end-user. The *Layout Manager* organizes the display based on the actual number of port, vehicle, and warehouse instances. *SystemClock* provides consistent time measurement to interested components, while the *Map* component informs vehicles of routes and distances. The *Telemetry* component determines when cargo arrives at a port, keeps track of available transport vehicles at each port, and tracks the cargo during its delivery to a warehouse. *TelemetryArtist* allows entry of new cargo as it arrives at a port and informs the *Telemetry* component when the end-user decides to route cargo. The *GraphicsBinding* component renders the drawing requests sent from the artists using a graphics toolkit, such as Java's AWT.

C2SADEL

We encountered a tension between formality and practicality in designing C2SADEL. Our goal was a language that was simple enough to be usable in practice, yet formal enough to adequately support analysis and evolution. For this reason, we kept the syntax simple and reduced formalism to a minimum.

A C2SADEL architecture is specified in three parts: component types, connector types, and topology. The topology, in turn, defines component and connector instances for a given system and their interconnections.[2] A partial description of the architecture shown in Fig. 3 is given in Fig. 4. The *DeliveryPort* component type is specified *extern*ally, i.e., in a different file (*Port.c2*). The *GraphicsBinding* component type

2. C2SADEL also supports hierarchical composition, allowing an entire architecture to be used as a single component in another architecture. We do not discuss this feature here due to space constraints. Its thorough treatment is given in [17].

```
architecture CargoRouteSystem is {
  component_types {
    component DeliveryPort is extern {Port.c2;}
    component GraphicsBinding is virtual {}
    ...
  }
  connector_types {
    connector FiltConn is {filter msg_filter;}
    connector RegConn is {filter no_filter;}
  }
  architectural_topology {
    component_instances {
      Runway : DeliveryPort;
      Binding : GraphicsBinding;
      ...
    }
    connector_instances {
      UtilityConn : FiltConn;
      BindingConn : RegConn;
      ...
    }
    connections {
      connector UtilityConn {
        top SimClock, DistanceCalc;
        bottom Runway, Truck;
      }
      connector BindingConn {
        top LayoutArtist, RouteArt;
        bottom Binding;
      }
      ...
    }
  }
}
```

Fig. 4. Cargo routing system architecture specified in C2SADEL.

is specified as a *virtual* type: it can be used in the definition of the topology, but it does not have a specification and does not affect type checking of the architecture; furthermore, a virtual type cannot be evolved via subtyping. The concept of virtual types is useful in the case of components, such as *GraphicsBinding*, for which implementations are known to already exist, but which are not specified in C2SADEL.

A partial specification of the *DeliveryPort* component is given in Fig. 5. Component invariants and operation pre- and postconditions in C2SADEL are specified as first-order logic expressions. *DeliveryPort*'s invariant specifies that the current capacity of the port will always be between zero and the maximum allowed capacity. Examples of a provided and a required operation are given. The required *ClockTick* operation should be provided by the *SystemClock* component in the *CargoRouteSystem* architecture, such that *DeliveryPort*'s variable placeholder *time* can be unified with the appropriate *SystemClock* state variable to satisfy the postcondition of the operation. As discussed in Section 2, *DeliveryPort*'s interface is separated from its behavior and a surjective function is provided to map between the two.

```
component DeliveryPort is
  subtype CargoRouteEntity (int \and beh) {
    state {
      cargo    : \set Shipment;
      selected : Integer;
      ...
    }
    invariant {
      (cap >= 0) \and (cap <= max_cap);
    }
    interface {
      prov ip_selshp: Select(sel : Integer);
      req  ir_clktck: ClockTick();
      ...
    }
    operations {
      prov op_selshp: {
        let  num : Integer;
        pre  num <= #cargo;
        post ~selected = num;
      }
      req or_clktck: {
        let  time : STATE_VARIABLE;
        post ~time = time + 1;
      }
      ...
    }
    map {
      ip_selshp -> op_selshp (sel -> num);
      ir_clktck -> or_clktck ();
      ...
    }
}
```

Fig. 5. DeliveryPort component type specified in C2SADEL. Interface and operation labels (e.g., ip_selshp) are a notational convenience. "~" denotes the value of a variable after the operation has been performed, while "#" denotes set cardinality.

The *DeliveryPort* component is a subtype of the more general *CargoRouteEntity* component, which is evolved by preserving both its interface and behavior, as shown in Fig. 5.

4 THE ENVIRONMENT

The concepts of component subtyping and type checking of architectures were initially motivated by specific problems encountered in exploring architecture-based development of distributed applications and OTS reuse in the context of the C2 style [18, 22, 37]. We have since developed a deeper understanding of many of the relevant ideas, including the formal underpinnings of the type theory [21]. To make the approach practical, we also needed to provide tool support for architecture-based subtyping and type checking, as well as for transferring the properties of an architecture to its implementation. This section describes the environment, called DRADEL (*D*evelopment of *R*obust *A*rchitectures using a *D*escription and *E*volution *L*anguage), which was developed to support modeling, analysis, evolution, and implementation of architectures described in C2SADEL.

The conceptual architecture of the DRADEL environment is shown in Fig. 6. Just like the application architectures it is built to support, the architecture of DRADEL itself adheres to C2 style rules. The environment is built using the C2 implementation infrastructure, an extensible framework of abstract classes for C2 concepts such as components, connectors, and messages [19]. The framework implements component interconnection and message passing protocols, and supports a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system process.[3]

Specifically, we have used the Java version of the C2 framework in the implementation of DRADEL. DRADEL's implementation consists of 13,000 source lines of code, in addition to the 7,000 lines of code in the base framework and C2 *GraphicsBinding*. In the current implementation, each DRADEL component executes in a separate thread of control within the same process.

DRADEL's Architecture

The *Repository* component from Fig. 6 stores architectures modeled in C2SADEL. The *Parser* receives via C2 messages specifications of architectures or sets of components, parses each specification, and requests that the *InternalConsistency-Checker* component check the consistency of the specification. The *InternalConsistencyChecker* builds its own representation of the architecture and ensures that components and

3. Note that implementation framework concepts, such as "abstract class," refer to the programming language used to implement an architecture, and are in no way related to our architectural type theory.

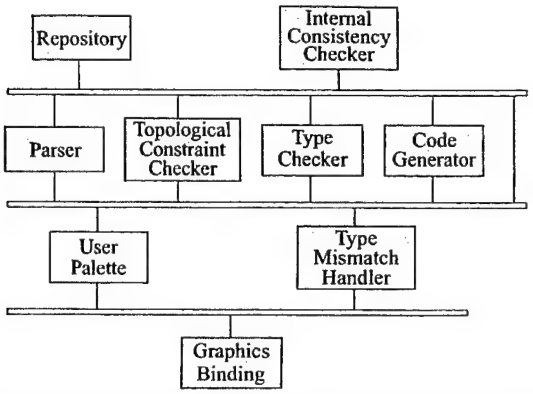


Fig. 6. Architecture of the DRADEL environment in the C2 style.

connectors are properly specified (e.g., two interface elements in a component cannot be identical), instantiated, and connected; that component interface elements are correctly mapped to operations (as specified in Section 2); that variables referenced in an operation are defined either as local variables for that operation or as component state variables; and that operation pre- and postcondition expressions are type correct (e.g., a set variable is never used in an arithmetic expression, although its cardinality may be). Furthermore, the *InternalConsistencyChecker* computes communication links for every component in an architecture: two components can interoperate if and only if they are on the opposite sides of the same connector (e.g., *Repository* and *Parser* in Fig. 6) or are on the opposite sides of two connectors which are, in turn, connected by one or more connector-to-connector links (e.g., *Repository* and *UserPalette* in Fig. 6).

Once the entire specification is parsed and its consistency ensured, its internal representation is broadcast to the *TopologicalConstraintChecker*, *TypeChecker*, and *CodeGenerator* components. If the specification in question is an architecture, the *TopologicalConstraintChecker* ensures that the topological rules of the C2 style are satisfied. The specification may contain a set of components instead, in which case no topological constraints are enforced.

The *TypeChecker* performs two kinds of analysis:

- given a specification of an architecture, it analyzes each component to establish whether its requirements are (partially or fully) satisfied by the components along its communication links;
- given a set of component specifications, the *TypeChecker* ensures that their specified subtyping relationships hold.

It performs these functions by establishing the relationships, discussed in Section 2, among component interface elements, invariants, and operations.

The *CodeGenerator* component generates application skeletons for the specified architecture or set of components. Like DRADEL itself, the application skeletons are built on top of the Java C2 framework. The "main program," containing component and connector instances and their interconnections, is automatically generated. For each specified component, the *CodeGenerator* creates the corresponding C2 component with the canonical internal architecture, shown in Fig. 7 [37]. The *InternalObject* of every generated component is a Java class

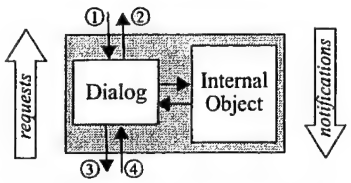


Fig. 7. Internal architecture of a canonical C2 component: (1) incoming notifications; (2) outgoing requests; (3) outgoing notifications; (4) incoming requests.

corresponding to the C2SADEL specification of the component. For example, the generated internal object for the *DeliveryPort* component from Fig. 5 is shown in Fig. 8: the state variables, state variable access methods, and provided component service declarations are generated.

Each method corresponding to a component service (e.g., *Select* in Fig. 8) is implemented as a null method in the generated class,[4] and is preceded by a comment containing the method's precondition and followed by one containing its postcondition. In general, these individual, application-specific methods are the only parts of a component for which the developers will have to provide an implementation. In the case of entirely new functionality, the pre- and postcondition comments serve as an implementation guideline to the developer; otherwise, they serve as an indicator of whether OTS functionality may be reused in the given context.

Except for the Java defined types (e.g., *Integer* or *String*), the *CodeGenerator* also supplies class skeletons for all other basic types, which include constructors and access methods. The

```
package c2.CargoRouteSystem;
import java.lang.*;
public class DeliveryPort extends Object
{
  // COMPONENT INVARIANT: cap >= 0 \and cap <= max_cap
  private Shipment_SET cargo;
  private Integer max_cap;
  private Integer cap;
  private Integer selected;

/***** Class Constructor *****/
  public DeliveryPort() {
    cargo = null;       // or: = new Shipment_SET(<init>);
    max_cap = null;     // or: = new Integer(<init>);
    cap = null;         // or: = new Integer(<init>);
    selected = null;    // or: = new Integer(<init>);
  }

/***** ADL Specified Methods *****/
  // PRECONDITION: num <= #cargo
  public void Select(Integer num)
  {
    /*** METHOD BODY ***/
  }
  // POSTCONDITION: ~selected = num

/***** State Variable Access Methods *****/
  public void SET_cargo(Shipment_SET new_value) {
    cargo = new_value;
  }
  public Shipment_SET GET_cargo() {
    return cargo;
  }
}
```

Fig. 8. Generated internal object class skeleton for the DeliveryPort component. The class is shown only partially due to space limitations.

4. The exception are non-void functions: Java requires their results to be initialized and returned. DRADEL initializes the results to an arbitrary value.

actual data structures must be specified in these classes by the developers. C2SADEL `set` types are currently implemented by subclassing from Java's *Vector* class, which provides a reasonable implementation of set abstractions.

The *Dialog* portion of a C2 component from Fig. 7 is responsible for all of component's message-based interaction. The *CodeGenerator* can generate a component's dialog almost completely from its C2SADEL specification: the provided services correspond to the notifications a component emits (message pathway 3 in Fig. 7) and the requests to which it is capable of responding (pathway 4), while the required services are used as a basis for specifying the requests the component issues (pathway 2). The dialog class also contains a specification of the component's message interface in the form needed by the underlying class framework to support various protocols of communication, e.g., message broadcast, registration, or point-to-point [37].

The only portion of the dialog that cannot be generated based on the information currently modeled in a C2SADEL specification is what the dialog should do in response to the notifications it receives (message pathway 1 in Fig. 7). This information could easily be specified in the ADL, as was shown in the prototype design language for C2-style architecture that preceded C2SADEL [24]; however, we have chosen to remove those constructs in the interest of language simplicity.

Given that a component's dialog is generated almost entirely from its C2SADEL specification, the internal object may, in fact, be completely replaced by an OTS component that does not communicate via messages. Essentially, the OTS component is modeled in C2SADEL, so that DRADEL can be used to check its compliance with the rest of the architecture and/or other components in its type hierarchy, as well as to generate its C2 dialog. This is the basic approach to OTS reuse we have used successfully in the past [19, 22].[5]

The remaining components in DRADEL's architecture, *UserPalette*, *TypeMismatchHandler*, and *GraphicsBinding* (see Fig. 6) handle the user's interaction with DRADEL. The *UserPalette* component drives the entire environment. It also displays the current execution status. The *TypeMismatch-Handler* component informs the user of the results of all component subtype matching and architectural type checking. Finally, as with any C2-style application, the *GraphicsBinding* renders DRADEL's user interface on the screen. The user interface is shown in Fig. 9, with examples of *nam*, *int*, and *beh* type conformance violations in the *CargoRouteSystem* architecture specified in Fig. 4.

The Process

The *UserPalette* component, rendered as the top pane of Fig. 9, enforces our "architecting" process. Prior to parsing a file, the *CheckConstr*, *TypeCheck*, and *GenCode* buttons are grayed out. Only once an architectural description or a set of components is successfully parsed and its internal consistency established can the user perform other functions. If the parsed file contains an architecture, the *CheckConstr* button is enabled, and the topological constraints must then be ensured. This must be done before either type checking the architecture or generating the application skeleton. If we are evolving design elements (i.e., ensuring the specified subtyping relationships among components), the *TypeCheck* and *GenCode* buttons are automatically enabled. This process is repeated every time the user decides to parse a new file.

5. Prior to DRADEL's development, OTS component dialogs were implemented manually.

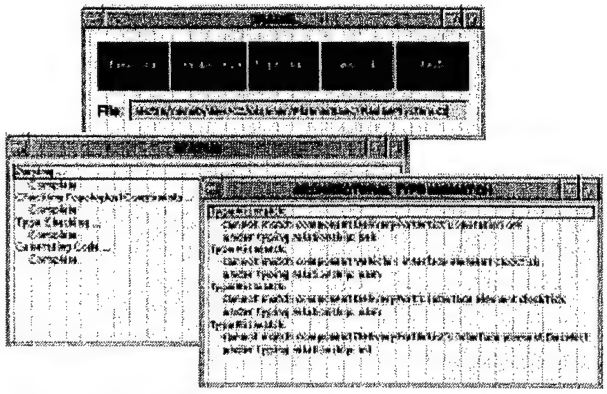


Fig. 9. DRADEL system's user interface.

If any errors are found during parsing, consistency checking, or topological constraint checking, a message will appear in the *Status* window informing the user of the exact error and its location. The user will not be able to proceed until the error is corrected. This is not the case with architectural type checking: the user can still generate the application, even if there are type errors, or the user may decide to skip the type checking stage altogether. The reasons for this are twofold.

Unlike a PL, which requires complete type conformance, architectures may describe meaningful functionality even if there are interface or behavioral mismatches. This has certainly been the case with C2-style applications, in which components communicate via asynchronous messages and are substrate independent [37]: C2 architectures are robust (hence the "R" in "DRADEL") in that a type mismatch may result in degraded functionality, or it may have no ill effects on the system, if it affects a part of the system that is not used in a given setting. It is the architect's responsibility to decide whether a given type mismatch is acceptable.

The other reason for allowing generation of type-mismatched components and architectures is the *TypeChecker* itself. To establish behavioral conformance between two components, the *TypeChecker* attempts to find mappings between their variables such that the correct (implication) relationships between their invariants, and pre- and postconditions hold. Since our approach does not explicitly model *basic types* (see Section 2), the *TypeChecker* essentially performs symbolic evaluation of logical expressions. For this reason, there are cases in which the *TypeChecker* is unable to correctly determine whether the implication indeed holds.

For example, assume that one component's operation precondition is

PRE1: `n <= 10`

and a candidate *beh*avior subtype's corresponding operation precondition is

PRE2: n^2 `<= 100`

where `n` in both expressions is declared to be of the (basic)

type *Natural*. To establish that the *beh*avior subtyping relationship holds, PRE1 must imply PRE2. This is obviously true: if a natural number is less than or equal to `10`, its square will be less than or equal to `100`. Note this would not be true of integers, for example (e.g., if `n = -20`). The *TypeChecker* has no knowledge of the non-negative property of natural numbers and cannot establish that the relationship between the two preconditions is true. It is pessimistically inaccurate: it treats those cases for which it cannot determine type conformance as errors. If the architect either judges the type mismatch not to be critical or discovers that the error is a result of *TypeChecker*'s inaccuracy, the architect has the choice to override the *TypeChecker* and proceed with code generation.

Discussion

DRADEL has been designed to be easily evolvable. Its components can be replaced to satisfy new requirements. For example, the file system-based *Repository* can be replaced with a database to keep track of design elements and architectures more efficiently. Another *Parser* can be substituted to support a different ADL, while a new *TopologicalConstraintChecker*, e.g., Armani [26], can be used to ensure adherence to a different architectural style. The *TypeChecker* may be replaced with a component that supports a different notion of architectural evolution and analysis; also, additional analysis tools may be added, even at runtime, using techniques described in [29]. Finally, the existing *CodeGenerator* may be replaced or new generation components added to support different implementation infrastructures.

DRADEL itself can be used reflexively to model and ensure the consistency of its own evolution. As the diagrams in Figures 3 and 6 and their subsequent discussions indicate, there are *no* fundamental differences between DRADEL and an application modeled, analyzed, evolved, and implemented with its help. Indeed, DRADEL's architecture can be specified in C2SADEL, parsed, checked for internal consistency, type checked, and the environment itself partially generated, as discussed above, using DRADEL.

5 RELATED WORK

This work has been influenced by research in several areas: OO typing, behavioral specification of software, software architectures and ADLs, and software environments. The relationship of our approach to OO typing and behavioral specification techniques is discussed in Section 2 and in more depth in [21]. In this section, we relate our approach to research in software architectures and environments. The unique aspects of our approach are

- component evolution via heterogeneous subtyping;
- formal specification, in type-theoretic terms, of the separation of provided and required functionality;
- analysis of architectures for consistency by unifying corresponding required and provided operations, where the architect possesses the authority to override the analysis tool ("architect's discretion");
- implementation generation; and
- a component-based, evolvable environment.

We look at related approaches mainly along these dimensions.

We have conducted an extensive survey of architectural approaches, which has indicated that their support for specification-time, architecture-based evolution is limited [20, 23]. ADLs that do address evolution typically rely on a chosen implementation language and only support a single form of subtyping/subclassing. Rapide [14] and LEDA [4] support evolution via inheritance. ACME [8] supports structural subtyping via its *extends* feature, while Aesop [7] supports behavior preserving subtyping to create substyles. UniCon [36] focuses on modeling, analyzing, and evolving software connectors; a connector is evolved by changing its properties or the properties of its interaction points (*roles*).

A majority of ADLs support some form of behavioral specification of components, connectors, and entire architectures. Three representative approaches are CHAM [12, 13], Rapide, and Wright [1]. These approaches allow the specification of dynamic behavior of components and/or connectors and the analysis of architectures for behavioral conformance (using the chemical abstract machine formalism, partially-ordered event sets and their patterns, and communicating sequential processes, respectively). Although, as discussed in Section 2, invariants and pre- and postconditions can be used to specify component interaction protocols, this has not been a particular focus or our research. Similarly to C2SADEL, the three approaches differentiate between provided and required functionality. CHAM and, to a limited extent, Rapide employ their equivalents of our variable placeholders to specify component expectations of the surrounding system. Wright addresses this issue indirectly, by interposing a connector between two, potentially independently developed, components. None of the approaches address the possibility of giving the architect the discretion to override a checking tool.

Few existing architectural approaches support mapping of architectures to their implementations. Some, like SRI's SADL [27], provide techniques for refining architectures across levels of abstraction, but do not carry the refinement into implementation. Others, like Rapide, provide a separate executable sublanguage in which systems may be implemented. Darwin [16], UniCon, and Weaves [10] generate "glue" code for composing architectural elements into a system, but assume that individual components will be independently implemented in a traditional PL. Our approach to generating an implementation from an architecture is influenced more directly by domain-specific software architecture (DSSA) approaches: by restricting the software development space to a specific architectural style, reference architecture, and possibly implementation infrastructure, DSSA projects, such as ADAGE [3] and MetaH [38], have been very successful at transferring architectural decisions to running systems. Aesop has been successful in this regard by adopting a similar philosophy: like C2, Aesop provides a class hierarchy for its concepts; this hierarchy, in essence, serves as a domain-specific language for implementing Aesop architectures.

Finally, our work on DRADEL has drawn inspiration from the Inscape environment for software specification, implementation, and evolution [33]. Inscape addressed many of the same issues addressed by DRADEL (scale, evolution, complexity, programming-in-the-large, practicality of the approach), but did so at a level of abstraction below architecture. For example, Inscape requires the semantics of data objects to be modeled, while DRADEL treats them as unelaborated (*basic*) types. Both approaches model operations

in terms of pre- and postconditions; Inscape also specifies *obligations*, predicates that must eventually be satisfied after an operation is invoked. Unlike our approach, which uses type-theoretic principles to evolve components, Inscape supports component evolution at a finer level of granularity and in a less systematic manner by adding, removing, and changing pre- and postcondition predicates, and also by changing the implementation itself. Inscape also provides component browsing, retrieval, and version management support. Although it has not been a focus of our work to date, we believe that DRADEL's architecture is flexible enough to include such functionality. Finally, it is unclear whether Inscape could be used reflexively in its own development and evolution.

6 CONCLUSIONS AND FUTURE WORK

Software architectures provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by focusing solely on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools that also must be supported with specific techniques to achieve desired properties. This paper has outlined such techniques and has discussed tool support for evolving software components in a manner that preserves the desired architectural properties and relationships.

These techniques are based on the recognition that software architectures need not always be rigid in establishing properties such as consistency and completeness. For example, it is not always the case that two components that share a communication link can actually communicate (e.g., due to mismatched interfaces). At the architectural level, this mismatch can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into the implemented system, implementation-time decisions may still allow the system to perform at least in a degraded mode. Thus, informing the architect of the potential problem and leaving the decision up to the architect is often preferable to automatically rejecting the option.

Architectures also have a great potential for improving the quality of the resulting software by allowing early analysis. Of course, ensuring specific properties of a system at the level of architecture is of little value unless it can also be ensured that those properties will be preserved in the resulting implementation. Attempting to do so manually is much more error prone than by utilizing a systematic and automated approach.

In this paper, we have presented a simple and practical approach for transferring architecture-level decisions to the implementation. We do not attempt to provide a general solution for system generation. That would essentially reduce architecture-based software development to a variant of transformational programming [32], thus inheriting all of the latter's problems and limitations, with the added problem of scale. Instead, the problem is rendered more tractable by focusing on a particular architectural style and implementation infrastructure. We do, however, have the flexibility to incrementally generalize our support for system generation since DRADEL is evolvable and can easily accommodate additional code generation tools.

DRADEL is a culmination of several years of research in software architectures and ADLs, evolution, and OTS reuse. We believe that the concepts embodied in DRADEL make it a promising candidate for extracting a more general, "reference" architecture for environments for architecture-based development and evolution and, in particular, for transferring architectural decisions to the implementation. We intend to investigate this possibility in the future.

A number of additional issues remain items of future work. We are currently considering several existing theorem provers and model checkers as possible complements to DRADEL's *TypeChecker*: NORA/HAMMR [35], Larch proof assistant (LP) [11], VCR [6], and PVS [30].[6] Operation pre- and postconditions are currently only comments in a generated implementation and serve as guidelines to developers. We intend to promote these comments to assertions that can be checked at system execution time, and are beginning to adapt the appropriate techniques for doing so [34]. Finally, we will fully integrate DRADEL with ArchStudio, our toolset for runtime architecture-based evolution [29]. Other items of future work include investigation of the applicability of our type theory for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, and automating the evolution of existing components to populate partial architectures.

ACKNOWLEDGEMENTS

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Approved for public release — distribution unlimited. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, Springer-Verlag, 1991.
3. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.
4. C. Canal, E. Pimentel, J. M. Troya. A pi-calculus Semantics for an Architecture Description Language. Technical Report,

6. Theorem provers and model checkers typically require some assistance from the user. This is entirely consistent with DRADEL's expectation that the user will play an active role in architecture-based system development and evolution.

LCC-ITI-98-07, Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain, April 1998.

5. K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.
6. B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-Based Software Component Retrieval Tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.
7. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, New Orleans, LA, USA, December 1994.
8. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
9. C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.
10. M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, Austin, TX, May 1991.
11. J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.
12. P. Inverardi and A. L. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, April 1995.
13. P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages (COORD '97)*, Berlin, 1997.
14. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
15. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.
16. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.
17. N. Medvidovic. Architecture-Based Specification-Time Software Evolution. Ph.D. Dissertation, University of California, Irvine, December 1998.
18. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.
19. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)* and *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.
20. N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.
21. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Type Theory for Software Architectures. Technical Report, UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.
22. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.
23. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.
24. N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium*, Los Angeles, CA, April 1996.
25. B. Meyer. Applying "Design by Contract." *IEEE Computer*, October 1992.
26. R. Monroe. Armani Language Reference Manual, version 0.1. Private communication, March 1998.
27. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, April 1995.
28. O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, Washington, D.C., USA, October 1993.
29. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.
30. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. *PVS:* Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, eds., *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, July/August 1996, Springer-Verlag.
31. J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, vol. 3, num. 2, 1992.
32. H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, September 1983.
33. D. E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, PA, May 1989.
34. D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, January 1995.
35. J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997.
36. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.
37. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
38. S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.
39. D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.
40. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, October 1997.

from the Proceedings of the First IFIP Working Conference on Software Architecture,
San Antonio, TX, Feb. 1999

Assessing the Suitability of a Standard Design Method for Modeling Software Architectures

Nenad Medvidovic and David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
{neno,dsr}@ics.uci.edu

Key words: Software architecture, architectural style, object-oriented design, architecture description languages, Unified Modeling Language

Abstract: Software architecture descriptions are high-level models of software systems. Most existing special-purpose architectural notations have a great deal of expressive power but are not well integrated with common development methods. Conversely, mainstream development methods are accessible to developers, but lack the semantics needed for extensive analysis. In our previous work, we described an approach to combining the advantages of these two ways of modeling architectures. While this approach suggested a practical strategy for bringing architectural modeling into wider use, it introduced specialized extensions to a standard modeling notation, which could also hamper wide adoption of the approach. This paper attempts to assess the suitability of a standard design method "as is" for modeling software architectures.

1. INTRODUCTION

Software architecture is an aspect of software engineering directed at reducing the costs of developing applications and increasing the potential for commonality among different members of a closely related product family

[6, 19]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so on [6, 7, 9, 19, 28, 29]. The basic promise of software architecture research is that better software systems can result from modeling their important aspects during, and especially early in the development. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research [13].

Part of the software architecture research community has focused on analytic evaluation of architecture descriptions. Many researchers have come to believe that, to obtain the benefits of an architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques [3, 5, 32]. Such languages are needed to demonstrate properties of a system upstream, thus minimizing the costs of errors. They are also needed to provide abstractions that are adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. A large number of architecture description languages (ADLs) has been proposed [2, 4, 9, 10, 12, 17, 25, 31].

Each ADL embodies a particular approach to the specification and evolution of an architecture. Answering specific evaluation questions demands powerful, specialized modeling and analysis techniques that address specific aspects in depth. However, the emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formal methods draws the modeler's attention away from day-to-day development concerns. The use of special-purpose modeling languages has made this part of the architecture community fairly fragmented, as revealed by a recent survey of architecture description languages [14].

Another part of the community has focused on modeling a wide range of issues that arise in software development, perhaps with a family of models that span and relate the issues of concern. By paying the cost of making such models, developers gain the benefit of clarifying and communicating their understanding of the system. However, emphasizing breadth over depth potentially allows many problems and errors to go undetected, because lack of rigor allows developers to ignore certain details. Several competing notations have been used in this part of the community, but there now exists a concerted effort to standardize methods for object-oriented analysis and design [18].

In our previous work, we described an approach to combining the advantages of specialized, highly formal methods of modeling architectures with general, less formal design methods [24]. This approach suggested a practical strategy for bringing architectural modeling into wider use, namely by incorporating substantial elements of architectural models into a standard design method, the Unified Modeling Language (UML) [20]. However, our technique is not without drawbacks: for each architectural approach and ADL, we introduced a somewhat specialized extension to UML. In particular, we relied heavily on UML's Object Constraint Language (OCL) [23] to specify architecture- and ADL-specific concepts.

OCL constraints are highly formal. Their formality may hamper wide adoption of our technique, although end users of the enhanced UML meta-model typically will not need to write OCL constraints. Furthermore, OCL is a part of the standard UML definition and it is expected that standardized UML tools will be able to process it. However, OCL is considered an uninterpreted part of UML and UML tools may not support it to the extent needed for creating, manipulating, analyzing, and evolving designs. For this reason, in this paper we attempt to assess the suitability of UML "as is" for modeling software architectures. In particular, we focus on one of the architectural approaches we addressed previously [24], the C2 architectural style [29]. We use a simple meeting scheduler application to highlight the issues. In the process, we attempt to shed light on the relationship between architecture and design.

The paper is organized as follows. The next section briefly describes UML. Section 3 briefly describes the example application, a meeting scheduler, used to illustrate our arguments throughout the paper. In Section 4, we introduce the C2 style and discuss a possible C2 architecture for the meeting scheduler application. Section 5 provides a "C2 style" UML design of the meeting scheduler. We discuss the results and key lessons learned in Section 6. Our conclusions round out the paper.

2. OVERVIEW OF UML

2.1 UML Background

A UML model of a software system consists of several partial models, each of which addresses a certain set of issues at a certain level of fidelity. There are eight issues addressed by UML models:

1. classes and their declared attributes, operations, and relationships;
2. the possible states and behavior of individual classes;

3. packages of classes and their dependencies;
4. example scenarios of system usage including kinds of users and relationships between user tasks;
5. the behavior of the overall system in the context of a usage scenario;
6. examples of object instances with actual attributes and relationships in the context of a scenario
7. examples of the actual behavior of interacting instances in the context of a scenario; and
8. the deployment and communication of software components on distributed hosts.

Fidelity refers to how close the model will be to the eventual implementation of the system: low-fidelity models tend to be used early in the life-cycle and are more problem-oriented and generic, whereas high-fidelity models tend to be used later and are more solution-oriented and specific. Increasing fidelity demands effort and knowledge to build more detailed models, but results in more properties of the model holding true in the system.

UML is a graphical language with fairly well-defined syntax and semantics. The syntax of the graphical presentation is specified by examples and a mapping from graphical elements to elements of the underlying semantic model [22]. The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints [21]. The meta-model is itself a UML model that specifies the abstract syntax of UML models. This is much like using a BNF grammar to specify the syntax of a programming language. For example, the UML meta-model states that a Class is one kind of model element with certain attributes, and that a Feature is another kind of model element with its own attributes, and that there is a one-to-many composition relationship between them.

UML is an extensible language in that new constructs may be added to address new issues in software development. Three mechanisms are provided to allow limited extension to new issues without changing the existing syntax or semantics of the language. (1) *Constraints* place semantic restrictions on particular design elements. (2) *Tagged values* allow new attributes to be added to particular elements of the model. (3) *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements. Another possible extension mechanism is to modify the meta-model, but this approach results in a completely new notation to which standard UML tools cannot be applied. We discuss this approach in more detail in Section 2.2.

Figure 1 shows the parts of the UML meta-model used in this paper. We have simplified the meta-model for purposes of illustration.

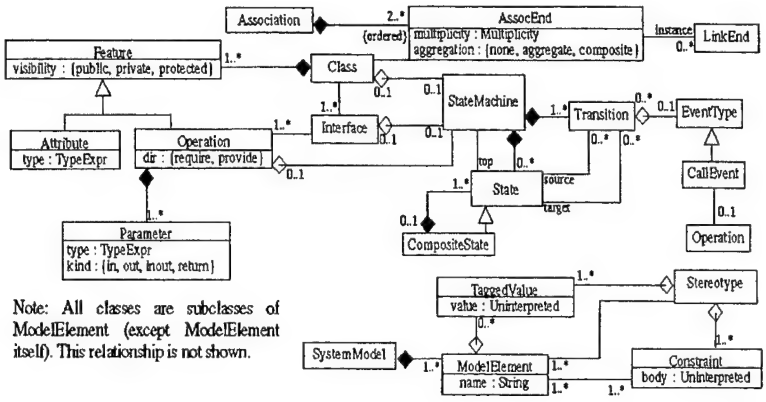


Figure 1. Simplified UML meta-model (adapted from [21]).

2.2 Our Strategy for Adapting UML for Architecture Modeling

In [24] we studied two possible approaches to using UML to model architectures. One approach is to define an ADL-specific meta-model. This approach has been used in more comprehensive formalizations of architectural styles [1, 12]. Defining a new meta-model helps to formalize the ADL, but does not aid integration with standard design methods. By defining our new meta-classes as subclasses of existing meta-classes we would achieve some integration. For example, defining Component as a subclass of meta-class Class would give it the ability to participate in any relationship in which Class can participate. This is basically the integration that we desire. However, this integration approach requires *modifications* to the meta-model that would not *conform* to the UML standard; therefore, we cannot expect UML-compliant tools to support it.

The approach for which we opted instead was to restrict ourselves to using UML's built-in extension mechanisms on existing meta-classes [24]. This allows the use of existing and future UML-compliant tools to represent the desired architectural models, and to support architectural style conformance checking when OCL-compliant tools become available. Our basic strategy was to first choose an existing meta-class from the UML meta-model that is semantically close to an ADL construct, and then define a

stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the ADL.

Neither of the two approaches answers the deeper question of UML's suitability for modeling software architectures "as is," i.e., without defining meta-models specific to a particular architectural approach or extending the existing UML meta-model. Such an exercise would highlight the respective advantages of special- and general-purpose design notations in modeling architectures. It also has the potential to further clarify the relationship between software architecture and design. Therefore, in this paper we study the characteristics of using the existing UML features to model architectures in a particular style, C2.

3. EXAMPLE APPLICATION

The example we selected to motivate the discussion in this paper is a simplified version of the meeting scheduler problem, initially proposed by van Lamsweerde and colleagues [8] and recently considered as a candidate model problem in software architectures [27]. We have chosen this problem partly because of our prior experience with designing and implementing a distributed meeting scheduler in the C2 architectural style, described in [29].

Meetings are typically arranged in the following way. A meeting initiator asks all potential meeting attendees for a set of dates on which they cannot attend the meeting (their "exclusion" set) and a set of dates on which they would prefer the meeting to take place (their "preference" set). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (the "date range").

The initiator also asks active participants to provide any special equipment requirements on the meeting location (e.g., overhead-projector, workstation, network connection, telephones); the initiator may also ask important participants to state preferences for the meeting location.

The proposed meeting date should belong to the stated date range and to none of the exclusion sets. It should also ideally belong to as many preference sets as possible. A date conflict occurs when no such date can be found. A conflict is strong when no date can be found within the date range and outside all exclusion sets; it is weak when dates can be found within the date range and outside all exclusion sets, but no date can be found at the intersection of all preference sets. Conflicts can be resolved in several ways:

- the initiator extends the date range;
- some participants expand their preference set or narrow down their exclusion set; or
- some participants withdraw from the meeting.

4. MODELING THE EXAMPLE APPLICATION IN C2

4.1 Overview of C2

C2 is a software architectural style for user interface intensive systems [29]. C2SADEL is an ADL for describing C2-style architectures [12, 15]; henceforth, in the interest of clarity, we use "C2" to refer to the combination C2 and C2SADEL. In a C2-style architecture, *connectors* transmit messages between components, while *components* maintain state, perform operations, and exchange messages with other components via two interfaces (named "top" and "bottom"). Each interface consists of a set of messages that may be sent and a set of messages that may be received. Inter-component messages are either *requests* for a component to perform an operation, or *notifications* that a given component has performed an operation or changed state.

In the C2 style, components may not directly exchange messages; they may only do so via connectors. Each component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. Request messages may only be sent "upward" through the architecture, and notification messages may only be sent "downward."

The C2 style further demands that components communicate with each other only through message-passing, never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure *substrate independence*, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language(s) in which the components or connectors are implemented, whether or not components execute in their own threads of control, the deployment of components to hosts, or the communication protocol(s) used by connectors.

4.2 Modeling the Meeting Scheduler in C2

Figure 2 shows a graphical depiction of a possible C2-style architecture for a simple meeting scheduler system. This system consists of components supporting the functionality of a *MeetingInitiator* and several potential meeting *Attendees* and *ImportantAttendees*. Three C2 connectors are used to

route messages among the components. Certain messages from the *Initiator* are sent both to *Attendees* and *ImportantAttendees*, while others (e.g., to obtain meeting location preferences) are only routed to *ImportantAttendees*. Since a C2 component has only one communication port on its top and one on its bottom, and all message routing functionality is relegated to connectors, it is the responsibility of *MainConn* to ensure that *AttConn* and *ImportantAttConn* above it receive only those message relevant to their respective attached components.

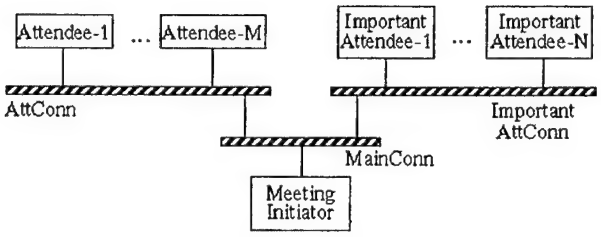


Figure 2. A C2-style architecture for a meeting scheduler system.

The *Initiator* component sends requests for meeting information to *Attendees* and *ImportantAttendees*. The two sets of components notify the *Initiator* component, which attempts to schedule a meeting and either requests that each potential attendee mark it in his/her calendar (if the meeting can be scheduled), or it sends other requests to attendees to extend the date range, remove a set of excluded dates, add preferred dates, or withdraw from the meeting. Each *Attendee* and *ImportantAttendee* component, in turn, notifies the *Initiator* of its date, equipment, and location preferences, as well as excluded dates. *Attendee* and *ImportantAttendee* components cannot make requests of the *MeetingInitiator* component, since they are above it in the architecture.

Most of this information is implicit in the graphical view of the architecture shown in Figure 2. For this reason, we specify the architecture in C2SADEL, a textual language for modeling C2-style architectures [11, 12, 15]. For simplicity, we assume that all attendees' equipment needs will be met, and that a meeting location will be available on the given date and that it will be satisfactory for all (or most) of the important attendees.

The *MeetingInitiator* component is specified below. The component only communicates with other parts of the architecture through its top port.

```
component MeetingInitiator is
    interface
        top_domain is
            out
                GetPrefSet ();
                GetExclSet ();
                GetEquipReqts ();
                GetLocPrefs ();
                RemoveExclSet ();
                RequestWithdrawal (to Attendee);
                RequestWithdrawal (to ImportantAttendee);
                AddPrefDates ();
                MarkMtg (d : date; l : loc_type);
            in
                PrefSet (p : date_rng);
                ExclSet (e : date_rng);
                EquipReqts (eq : equip_type);
                LocPref (l : loc_type);
        bottom_domain is
            out null;
            in null;
    parameters null;
    methods
        procedure Start ();
        procedure Finish ();
        procedure SchedMtg (p : set date_rng; e : set date_rng);
        procedure AddPrefSet (pref : date_rng);
        procedure AddExclSet (exc : date_rng);
        procedure AddEquipReqts (eq : equip_type);
        procedure AddLocPref (l : loc_type);
        function AttendInfoCompl () return boolean;
        procedure IncNumAttends (n : integer);
        function GetNumAttends () : return integer;
    behavior
        startup
            invoke_methods Start;
            always_generate GetPrefSet, GetExclSet, GetEquipReqts,
GetLocPrefs;
        cleanup
            invoke_methods Finish;
            always_generate null;
        received_messages PrefSet;
            invoke_methods AddPrefSet, IncNumAttends, AttendInfoCompl,
GetNumAttends, SchedMtg;
            may_generate RemoveExclSet xor RequestWithdrawal xor
MarkMtg;
        received_messages ExclSet;
            invoke_methods AddExclSet, AttendInfoCompl, GetNumAttends,
SchedMtg;
            may_generate AddPrefDates xor RemoveExclSet xor
RequestWithdrawal xor MarkMtg;
        received_messages EquipReqts;
            invoke_methods AddEquipReqts, AttendInfoCompl,
GetNumAttends, SchedMtg;
            may_generate AddPrefDates xor RemoveExclSet xor
RequestWithdrawal xor MarkMtg;
        received_messages LocPref;
            invoke_methods AddLocPref;
```

```
            always_generate null;
    context
        bottom_most computational_unit;
end MeetingInitiator;
```

The *Attendee* and *ImportantAttendee* components receive meeting scheduling requests from the *Initiator* and notify it of the appropriate information. The two types of components only communicate with other parts of the architecture through their bottom ports.

```
component Attendee is
    interface
        top_domain is
            out null;
            in null;
        bottom_domain is
            out
                PrefSet (p : date_rng);
                ExclSet (e : date_rng);
                EquipReqts (eq : equip_type);
                Witdrawn ();
            in
                GetPrefSet ();
                GetExclSet ();
                GetEquipReqts ();
                RemoveExclSet ();
                RequestWithdrawal ();
                AddPrefDates ();
                MarkMtg (d : date; l : loc_type);
    parameters null;
    methods
        procedure Start ();
        procedure Finish ();
        procedure NoteMtg (d : date; l : loc_type);
        function DeterminePrefSet () return date_rng;
        function DetermineExclSet () return date_rng;
        function AddPrefDates () return date_rng;
        function RemoveExclSet () return date_rng;
        procedure DetermineEquipReqts (eq : equip_type);
    behavior
        startup
            invoke_methods Start;
            always_generate null;
        cleanup
            invoke_methods Finish;
            always_generate null;
        received_messages GetPrefSet;
            invoke_methods DeterminePrefSet;
            always_generate PrefSet;
        received_messages AddPrefDates;
            invoke_methods AddPrefDates;
            always_generate PrefSet;
        received_messages GetExclSet;
            invoke_methods DetermineExclSet;
            always_generate ExclSet;
        received_messages GetEquipReqts;
```

```
            invoke_methods DetermineEquipReqts;
            always_generate EquipReqts;
        received_messages RemoveExclSet;
            invoke_methods RemoveExclSet;
            always_generate ExclSet;
        received_messages RequestWithdrawal;
            invoke_methods Finish;
            always_generate Withdrawn;
        received_messages MarkMtg;
            invoke_methods NoteMtg;
            always_generate null;
    context
        top_most computational_unit;
end Attendee;
```

ImportantAttendee is a specialization of the *Attendee* component: it duplicates all of *Attendee*'s functionality and adds specification of meeting location preferences. *ImportantAttendee* is thus specified as a subtype of *Attendee* that preserves its interface and behavior, but can implement that behavior in a new manner.

```
component ImportantAttendee is subtype Attendee (int and beh)
    interface
        bottom_domain is
            out
                LocPrefs (l : loc_type);
            in
                GetLocPrefs ();
    methods
        function DetermineLocPrefs () return loc_type;
    behavior
        received_messages GetLocPrefs;
            invoke_methods DetermineLocPrefs;
            always_generate LocPrefs;
end ImportantAttendee;
```

The *MeetingScheduler* architecture depicted in Figure 2 is shown below. The architecture is specified with conceptual components (i.e., component types). Each conceptual component (e.g., *Attendee*) can be instantiated multiple times in a system.

```
architecture MeetingScheduler is
    conceptual_components
        top_most
            Attendee;
            ImportantAttendee;
        internal null
        bottom_most
            MeetingInitiator;
    connectors
        connector MainConn is
            message_filter no_filtering;
        end MainConn;
```

```
        connector AttConn is
            message_filter no_filtering;
        end AttConn;
        connector ImportantAttConn is
            message_filter no_filtering;
        end ImportantAttConn;
    architectural_topology
        connector AttConn connections
            top_ports
                Attendee;
            bottom_ports
                MainConn;
        connector ImportantAttConn connections
            top_ports
                ImportantAttendee;
            bottom_ports
                MainConn;
        connector MainConn connections
            top_ports
                AttConn;
                ImportantAttConn;
            bottom_ports
                MeetingInitiator;
end MeetingScheduler;
```

An instance of the architecture (a system) is specified by instantiating the components. For example, an instance of the meeting scheduler application with three participants and two important participants is specified as follows.

```
system MeetingScheduler_1 is
    architecture MeetingScheduler with
            Attendee instance Att_1, Att_2, Att_3;
            ImportantAttendee instance ImpAtt_1, ImpAtt_2;
            MeetingInitiator Instance MtgInit_1;
end MeetingScheduler_1;
```

5. MODELING THE C2-STYLE MEETING SCHEDULER APPLICATION IN UML

The process of designing a C2-style application in UML should be driven and constrained both by the rules of C2 and the modeling features available in UML. The two must be considered simultaneously. For this reason, the initial steps in this process are to develop a domain model for a given application in UML and an informal C2 architectural diagram, such as the one from Figure 2. Such an architectural diagram is key to making the appropriate mappings between classes in the domain and architectural components. Furthermore, it points to the need to explicitly model connectors in any C2-style architecture. Another important aspect of C2

architectures is the prominence of components' message interfaces. This is reflected in a UML design by modeling interfaces explicitly and independently of the classes that will implement those interfaces.

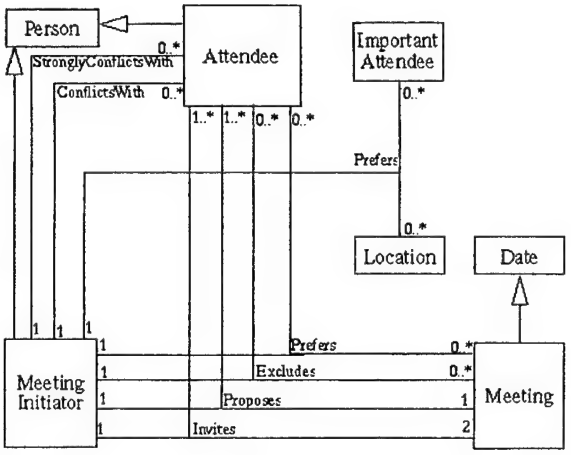


Figure 3. UML class diagram for the meeting scheduler application. Details (attributes and methods) of each individual class have been suppressed for clarity.

Our initial attempt at a UML class diagram for the meeting scheduler application is shown in Figure 3. The diagram shows the domain model for the meeting scheduler application consisting of the domain classes, their inheritance relationships, and their associations. The diagram abstracts away many architectural details, such as the mapping of classes in the domain to implementation components, the order of interactions among the different classes, and so forth. Furthermore, much of the semantics of class interaction is missing from the diagram. For example, the *Invites* association associates two *Meetings* with one or more *Attendees* and one *MeetingInitiator*. However, the association does not make clear the fact that the two *Meetings* are intended to represent a range of possible meeting dates, rather than a pair of related meetings.

Each class exports one or more interfaces, shown in Figure 4. The *ImportantMtgInit* and *ImportantMtgAttend* interfaces inherit from the *MtgInit* and *MtgAttend* interfaces, respectively. The only difference is the added operation to request and notify of location preferences.

Note that every interface element corresponds to a C2 message in the architecture specified in Section 4.2. All methods in the UML design will be implemented as asynchronous message passes, as they would in C2. Since C2 components communicate via implicit invocation, C2 messages do not have return values; this is also reflected in Figure 4.

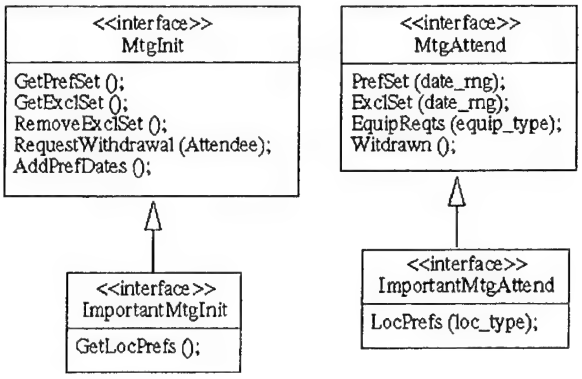


Figure 4. Meeting scheduler class interfaces.

In order to model a C2 architecture in UML, connectors must be defined. Although connectors fulfill a role different from components, they can also be modeled with UML classes. However a C2 connector is by definition generic and can accommodate any number ant type of C2 components; informally, the interface of a C2 connector is a union of the interfaces of its attached components. UML does not support this form of genericity, so that the connectors specified in UML have to be application-specific. For that purpose, the connectors for the meeting scheduler application share the components' interfaces. Each connector can be thought of as a simple class that forwards each message it receives to the appropriate components. Therefore, while the component class interface specifications, shown in Figure 4, correspond to the different C2 components' outgoing messages (i.e., their provided functionality), the connector interfaces are routers of both the incoming and outgoing messages, as depicted in Figure 5. Connectors do not add any functionality at the domain model level; we have thus chosen to omit them from the class diagram in Figure 3.

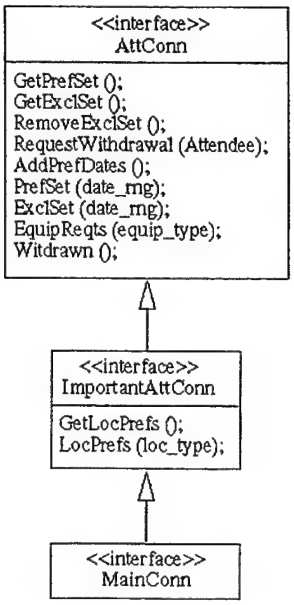


Figure 5. Application-specific UML classes representing C2 connectors.

A refined class diagram for the meeting scheduler application is shown in Figure 6. The *Attendee* and *ImportantAttendee* classes are related by interface inheritance, which is depicted in Figure 4, but is only implicit in Figure 6 (and altogether omitted from Figure 3). We have omitted from Figure 6 the *Location*, *Meeting*, and *Date* classes shown in Figure 3, since they have not been impacted. We have also omitted the two superclasses for the components and connectors (*Person* and *Conn*, respectively).

Note that the class diagram in Figure 6 is similar in its structure to the C2 architecture depicted in Figure 2. The only difference is that the diagram in Figure 2 depicts instances of the different components and connectors, while a UML class diagram depicts classes and their associations. UML provides several types of diagrams that depict class instances (objects). One candidate is UML's object diagrams; however, we choose to depict a collaboration diagram to further draw the contrast between UML and C2.

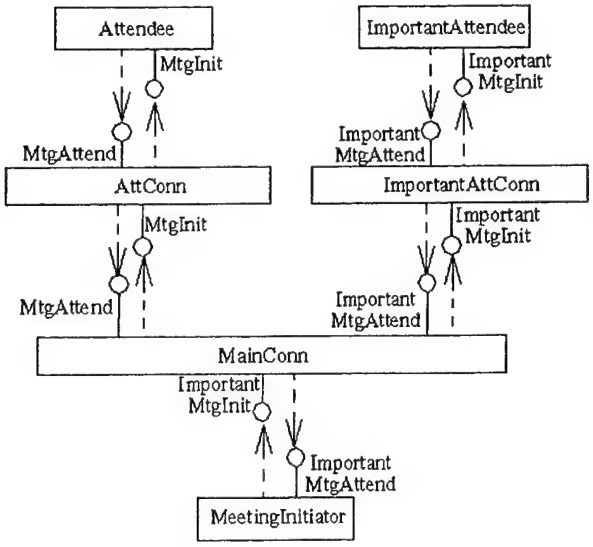


Figure 6. UML class diagram for the meeting scheduler application designed in the C2 architectural style.

Figure 7 shows the collaboration between an instance of the *MeetingInitiator* class (*MI*) and any instances of *Attendee* and *ImportantAttendee* classes: *MI* issues a request for a set of preferred meeting dates; *MC*, an instance of the *MainConn* class routes the request to instances of both connectors above it, *AC* and *IAC*, which, in turn, route the requests to all components attached on their top sides; each participant component chooses a preferred date and notifies any components below it of that choice; these notification messages will eventually be routed to *MI* via the connectors. Note that, if *MI* had sent the request to get meeting location preferences (*GetLocPrefs* in the *ImportantMtgInit* interface in Figure 4), *MC* would have routed them only to *IAC* and none of the instances of the *Attendee* class would have received that request

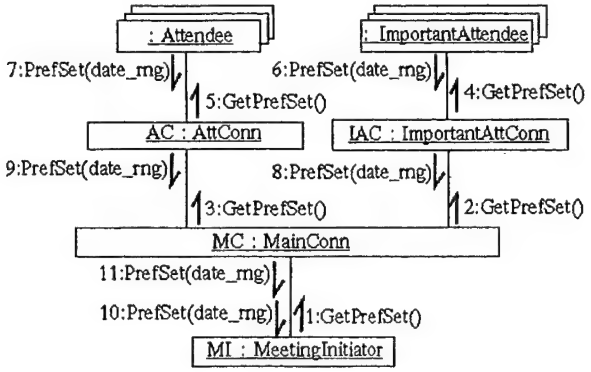


Figure 7. Collaboration diagram for the meeting scheduler application showing a response to a request issued by the *MeetingInitiator* to both *Attendees* and *ImportantAttendees*.

The above diagrams, and particularly Figure 6, differ from a C2 architecture in that they explicitly specify only the messages a component receives (via interface attachments to a component rectangle). UML also allows specification of messages a component sends; we believe those messages to be obvious from the diagram and have thus chosen to omit them to simplify the diagrams.

6. DISCUSSION

The exercise of modeling a C2-style architecture in UML has been fairly successful. Part of the success can be attributed to the fact that many architectural concepts are found in UML (e.g., interfaces, component associations, behavioral modeling, and so forth). On the other hand, the modeling capabilities provided by UML do not always fully satisfy the needs of architectural description. We discuss several major similarities and differences in this section.

6.1 Software Modeling Philosophies

Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language or thread of control. C2 limits communication to asynchronous message

passing and UML supports this restriction. Both C2 and UML include specifications of messages that may be sent and received.

Although we did not model details of the internal parts of a C2 component or the behavior of any C2 constructs (components, connectors, communication ports, and so forth) in our UML specification, we believe that many of those aspects could be modeled with UML's sequence, collaboration, statechart, and activity diagrams. Existing ADLs, including C2SADEL, are often not able to support all of these kinds of semantic models [14].

6.2 Assumptions

Like any notation, UML embodies its developers' assumptions about its intended usage. "Architecting" a system was not an intended use of UML. While one can indeed focus on the different perspectives when modeling a system (discussed above), a software architect may find that the support for those perspectives found in UML only partially satisfies his/her needs.

For example, in modeling the collaboration among C2 components shown in Figure 7, we were forced to assign a relative ordering to messages in the architecture. In effect, since all C2 components and connectors can execute in their own thread(s) of control, such an ordering cannot always be determined. Indeed, it is possible that message 4 would be sent before message 3.

6.3 Problem Domain Modeling

UML supports modeling a problem domain, as we have briefly shown in this paper. A C2 architectural model, however, often hides some of the information present in a domain model. For example, meeting, equipment, and location information is present in Figure 3, but is missing from the C2 architecture specified in Section 4 and its corresponding UML diagram in Figure 6. Modeling all the relevant information early in the development lifecycle is crucial to the success of a software project. Therefore, a domain model should be considered a separate and useful architectural perspective [13, 30].

6.4 Architectural Abstractions

Some concepts of C2, and software architectures in general, are very different from those of UML and object-oriented design in general. Connectors are first-class entities in C2. While the functionality of a connector can typically be abstracted by a class/component [9, 10], C2

connectors have the added property that their interfaces are context reflective. This property is designed into C2SADEL and C2's implementation infrastructure [16] for all connectors, whereas the approach described in this paper requires specialized modeling of application-specific connector classes in UML.

The underlying problem is even deeper. Although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides may not match an architect's mental model of the system as faithfully as the architect's ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices [26]. We believe this to be a key issue and one that argues against considering a notation like UML to be a "mainstream" ADL: a given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect's job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

6.5 Architectural Styles

Architecture is the appropriate level of abstraction at which rules of a compositional style (i.e., an architectural style) can be exploited and should be elaborated. Doing so results in a set of heuristics that, if followed, will guarantee a resulting system certain desirable properties.

Standard UML provides no support for architectural styles. The rules of different styles have to be built into UML by constraining its meta-model, as we have done previously [24]. Therefore, in choosing to use UML "as is", we have removed one shortcoming of our previous approach, only to introduce another. In particular, every C2 architecture designed in the manner we described in this paper adheres to the UML meta-model and, as such, can be understood by a typical UML user and manipulated with standardized UML tools. On the other hand, the process of modeling a C2 architecture in UML is heuristic- rather than constraint-driven. Therefore, there is no guarantee that the designer will always adhere to the rules of C2. For this reason, it may also be more difficult to provide support for automated translation of "C2-style" UML designs into C2SADEL for C2-specific manipulations.

7. CONCLUSIONS

We found this initial attempt at modeling a C2-style architecture in UML useful. It highlighted those UML characteristics that show potential for aiding architectural modeling, but also pointed out some of UML's shortcomings in this regard. This experience can also serve as a solid basis for further study, both with other C2 architectures, as well as with other ADLs (e.g., Wright [2]) and architectural styles (e.g., client-server).

Before we can draw definitive conclusions about the relative merits of this approach and the approach described in our previous work [24], further research into the techniques described in the two papers is needed. One necessary step to integrate UML with other ADLs discussed in [24]: Wright [2], Darwin [10], and Rapide [9]. Each of these ADLs has certain aspects in common with UML; these were expressed with UML's extension mechanisms. We intend to investigate whether they can also be expressed in UML without extensions.

Our experience to date indicates that adapting UML to address architectural concerns requires reasonable effort, has the potential to be a useful complement to ADLs and their analysis tools, and may be a practical step toward mainstream architectural modeling. Using UML has the benefits of leveraging mainstream tools, skills, and processes. It may also aid in the comparison of ADLs because it forces some implicit assumptions to be explicitly stated in common terms.

ACKNOWLEDGEMENTS

We wish to thank J. Robbins and D. Redmiles for their insights into the issues in integrating UML with architectures and their collaboration on other aspects of this work.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced

Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

1. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, pp. 319-364, October 1995.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, pp. 213-249, July 1997.
3. D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.
4. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pp. 175-188, New Orleans, Louisiana, USA, December 1994.
5. D. Garlan, F. N. Paulisch, and W. F. Tichy, editors. Summary of the Dagstuhl Workshop on Software Architecture, February 1995. Reprinted in ACM Software Engineering Notes, pp. 63-83, July 1995.
6. D. Garlan and M. Shaw. *An introduction to software architecture: Advances in software engineering and knowledge engineering*, volume I. World Scientific Publishing, 1993.
7. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pp. 42-50, November 1995.
8. A. van Lamsweerde, R. Darimont and P. Massonet. The Meeting Scheduler System: Preliminary Definition. University of Louvain, Unite d'informatique, B-1348 Louvain-la-Neuve (Belgium), October 1992.
9. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pp. 717-734, September 1995.
10. J. Magee and J. Kramer. Dynamic structures in software architecture. *In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 3-14, San Francisco, CA, October 1996.
11. N. Medvidovic. ADLs and Dynamic Architecture Changes. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 24-27, San Francisco, CA, October 1996.
12. N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pp. 28-40, Los Angeles, CA, April 1996.
13. N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain Specific Languages*, pp. 199-212, Santa Barbara, CA, October. 1997.
14. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 22-25, 1997.
15. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 24-32, San Francisco, CA, October 1996.

16. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pp. 190-198, Boston, MA, May 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pp. 692-700, Boston, MA, May 1997.
17. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pp. 356-372, April 1995.
18. Object Management Group. Object analysis and design RFP-1. Object Management Group document ad/96-05-01. June 1996. Available from http://www.omg.org/docs/ad/96-05-01.pdf.
19. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pp. 40-52, October 1992.
20. Rational Partners (Rational, IBM, HP, Unisys, MCI, Microsoft, ObjecTime, Oracle, i-Logix, etc.). Proposal to the OMG in response to OA&D RFP-1. Object Management Group document ad/97-07-03. July 1997. Available from http://www.omg.org/docs/ad/.
21. Rational Partners. UML Semantics. Object Management Group document ad/97-08-04. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-04.pdf.
22. Rational Partners. UML Notation Guide. Object Management Group document ad/97-08-05. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-05.pdf.
23. Rational Software Corporation and IBM. Object constraint language specification. Object Management Group document ad/97-08-08. September 1997. Available from http://www.omg.org/docs/ad/.
24. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 209-218, Kyoto, Japan, April 19-25, 1998.
25. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pp. 314-335, April 1995.
26. M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Springer-Verlag Lecture Notes in Computer Science, Volume 1000, 1995.
27. M. Shaw, D. Garlan, R. Allen, D. Klein, J. Ockerbloom, C. Scott, M. Schumacher. Candidate Model Problems in Software Architecture. Unpublished manuscript, November 1995. Available from http://www.cs.cmu.edu/afs/cs/project/compose/www/html/ModProb/.
28. D. Soni, R. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering*, pp. 196-207, Seattle, WA, April 1995.
29. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pp. 390-406, June 1996.
30. W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, July 1995.
31. S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.
32. A. L. Wolf, editor. Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Francisco, CA, October 1996.

from the Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, May 16–22, 1999

Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures

Elisabetta Di Nitto
CEFRIEL - Politecnico di Milano
Via Fucini, 2
20133 Milano, Italy
+39 2 23954 272
dinitto@elet.polimi.it

David Rosenblum
University of California, Irvine
Dept. of Information and Computer Science
Irvine, CA 92697-3425 USA
+1 949 824 6534
dsr@ics.uci.edu

ABSTRACT
Architecture Definition Languages (ADLs) enable the formalization of the architecture of software systems and the execution of preliminary analyses on them. These analyses aim at supporting the identification and solution of design problems in the early stages of software development. We have used ADLs to describe *middleware-induced architectural styles*. These styles describe the assumptions and constraints that middleware infrastructures impose on the architecture of systems. Our work originates from the belief that the explicit representation of these styles at the architectural level can guide designers in the definition of an architecture compliant with a pre-selected middleware infrastructure, or, conversely can support designers in the identification of the most suitable middleware infrastructure for a specific architecture.

In this paper we provide an evaluation of ADLs as to their suitability for defining middleware-induced architectural styles. We identify new requirements for ADLs, and we highlight the importance of existing capabilities. Although our experimentation starts from an attempt to solve a specific problem, the results we have obtained provide general lessons about ADLs, learned from defining the architecture of existing, complex, distributed, running systems.

Keywords
Architectural styles, architecture definition languages, event-based interaction, middleware infrastructures, software architectures

1 INTRODUCTION

The development of complex systems demands well established approaches that facilitate robustness of products, economy of the development process, and rapid time to market. This need has led, in the last few years, to the establishment of a research area called *software architecture* [20, 7]. In this area, researchers have demonstrated the usefulness of formalizing the definition of the high-level structure of software systems and allowing designers to perform preliminary analysis on the system under development. Such analysis aims at discovering and solving design problems in the early stages of development. To support the definition of software architectures, a number of *Architecture Definition Languages* (ADLs) [12] have been proposed. Also, a number of *architectural styles* are being identified [21]. A style defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. New architectures can be defined as instances of specific styles.

While these contributions hold the promise of setting up a formal foundation for the definition of software architectures, others are taking a more pragmatic approach to the development of distributed systems and are focusing on the definition of *middleware infrastructures* (or *middleware* for short), such as ActiveX/DCOM, CORBA, and Enterprise JavaBeans [19]. These infrastructures support the development of applications composed of several, possibly distributed, components and provide mechanisms to enable communication among components and to hide their distribution. Also, they offer a number of *predefined components* that provide well-defined classes of operations. For instance, CORBA defines a set of service components that support transactional communication, event-based interaction, security, etc. [17].

We argue that, despite the fact that architectures and middleware address different phases of software development, the usage of middleware and predefined components can influence the architecture of the system being developed. Conversely, specific architectural choices constrain the selection of the underlying middleware used in the implementation phase. A similar observation has been presented in [5], in which the *architectural mismatches* generated by the assumptions reusable parts make about the architecture of an application are identified.

For a system to be implemented in a straightforward manner on top of a middleware, the corresponding architecture has to be compliant with the architectural constraints imposed by the middleware. Sullivan et al. in [23] corroborate this

ICSE '99 Los Angeles CA

claim by demonstrating that a style, that in principle seems to be easily implementable using the COM middleware, is actually incompatible with it. This view has been stated also in [18], in which the authors discuss the importance of complementing *component interoperability models* with explicit architectural models. Regis [9] and C2 [24] are middlewares for which ADLs have been specifically defined (Darwin [10] and C2SADEL [13], respectively). These ADLs support the definition of architectures compliant with the corresponding middleware. From a different viewpoint, the ADL UniCon [22] proposes a conceptually similar approach. It predefines in the language a set of connectors that have an associated implementation. These connectors support the definition of an architecture and are part of the implementation of the corresponding system.

While Regis, C2, and UniCon define an architectural definition environment strictly tied to the implementation environment, we aim at developing a more general approach. In particular, we aim to capture the architectural assumptions induced by middlewares in terms of *middleware-induced styles*. In essence, we say that a class of related forms of middleware induces the definition of an architectural style, with each specific middleware of the class defining a *variation* of that style. We use a number of general-purpose ADLs to describe these styles and variations. Our attempt aims at demonstrating that the explicit availability of middleware-induced styles is extremely useful in guiding the architect in the definition of the architecture of an application and in selecting the most suitable middleware, independently of any special purpose development environment.

Unfortunately, our experience in using ADLs has not been fully satisfying. In particular, many available ADLs themselves introduce specific architectural assumptions, which can conflict with the ones embodied in existing middleware. In this paper we discuss our experience in using ADLs to define middleware-induced styles. We provide an evaluation of the ADLs we used, identify new requirements, and highlight the importance of existing capabilities. Although our experimentation starts from an attempt to solve a specific problem, the results we have obtained provide general lessons about ADLs, learned from defining the architecture of existing, complex, distributed, running systems.

The rest of the paper is structured as follows: Section 2 provides a brief introduction to ADLs. Section 3 presents two *event-based middlewares* that we have selected for our case studies. Section 4 describes our experience in using existing ADLs to specify the styles implemented by these middlewares. Section 5 provides an evaluation of our experience and summarizes the requirements we expect from ADLs. Section 6 provides some conclusions and discusses future work.

2 MAIN CHARACTERISTICS OF ADLs

ADLs have been thoroughly surveyed and classified in [12]. In this section we present their salient features.

ADLs usually define three main entities as primitive concepts: *components*, *connectors*, and *configurations*. A component represents a unit of computation and interacts with other components through connectors. A configuration represents the way components and connectors are composed to define a specific architecture. These basic concepts are interpreted by different ADLs in different ways. This variety in the expressive power of ADLs can make descriptions of the same architecture in different languages substantially different. In general, these differences are symptoms of a more serious disagreement on what architectural descriptions should express and what the right level of abstraction is for them.

For the purposes of our work, a middleware-induced style can be specified by describing the "characteristics" that components, connectors, and configurations of instances of the style must have to be compliant with the corresponding middleware. The characteristics to be represented have been selected according to our experience with different middleware. As we will discuss in the following sections, these characteristics encompass dynamic behavioral properties, constraints on single components or connectors or on the way they interact, and topological constraints on the way components and connectors are attached to compose a system.

3 CASE STUDIES

We chose as case studies two representative middlewares of the event-based paradigm: JEDI [3] and C2 [24]. In general, event-based middlewares support the implementation of systems in which components communicate by generating and receiving *events*. Events *published* by a source are *notified* to all components that have declared an interest in receiving them. The middlewares themselves take care of event observation and notification, thus guaranteeing a complete decoupling between sources and recipients of events.

The two case studies present interesting and complementary characteristics. JEDI is a typical representative of the event-based middleware category (including technologies such as the CORBA Event Service). The style it implies is easily generalizable and tailorable for other event-based middlewares. C2 imposes several constraints on the topology of architectures. In the following, we informally describe these two middlewares by highlighting the capabilities they offer to application developers and the rules they impose.

The C2 Runtime Infrastructure

The C2 middleware provides an object-oriented class framework and implements a specific architectural style that constrains the design of applications [24]. C2 was initially created to support the flexible development of GUI-based systems, but it has proven to be suitable as a general-purpose middleware.

As we have mentioned in the introduction, the C2 develop-

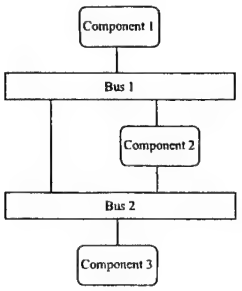


Figure 1: An example of system structure in C2.

ment environment is currently being enriched with a special-purpose ADL that supports the definition of architectures compliant with the C2 style [13]. However, it is still possible to implement a C2 system by using the framework provided by the middleware independently of the architectural definition environment. In the context of this paper we disregard this architecture definition environment. Our goal, in fact, is to assess the ability of ADLs developed independently of C2 to represent the style C2 endorses.

In the C2 style an application is composed of components that communicate through *buses* (see Figure 1). The communication is event-based. In particular, a component can send (receive) events to (from) the buses to which it is attached. Buses are in charge of delivering these events to components according to a policy that will be explained later on. Components and buses can be composed in several ways, provided that the following *topological rules and constraints* hold:[1]

1. Each component and bus has two connection points, one called *top* and the other called *bottom*.
2. The top (bottom) of a component can be attached to the bottom (top) of only one bus.
3. It is not possible for components to be attached directly to each other; buses always have to act as intermediaries between them.
4. Conversely, buses can be attached together. In this case each bus considers the other as a component as far as the publication and forwarding of events is concerned.
5. It is not possible to attach the top (bottom) of a component (or of a bus) to the top (bottom) of a bus.
6. It is not possible to have cycles. That is, a component can never receive a notification generated by itself. The meaning of this constraint will be clarified by the definition of the behavior of a C2 architecture given below.

The *behavior* of components and buses can be summarized as follows: A component can send *request* events to the bus attached to its top (if any). Also, it can send *notification* events to the bus attached to its bottom (if any). When a bus receives a request from its bottom, it forwards this request to all the components (and buses) attached to its top that can handle this request.[2] When a bus receives a notification from its top, it forwards this notification to all the components (and buses) that are attached to its bottom.

[1] The rationale for these rules is presented in [24].

JEDI

JEDI [3] is an event-based middleware developed to support lightweight and decoupled communication among distributed components called *active objects*. An active object can dynamically *subscribe/unsubscribe* to events and *publish* them. An *event dispatcher* stores the subscriptions in its internal state. Moreover, it forwards published event notifications to all the active objects that have previously issued a subscription matching these notifications. When the event dispatcher receives an unsubscription, it deletes the corresponding subscription. An active object can temporarily disconnect (*move-out*) from the event dispatcher and reconnect (*move-in*), either from the same location or from another location. The event dispatcher keeps track of all the notifications directed to a temporarily disconnected active object and delivers them when the active object reconnects again. The only constraint imposed on the topology of architectures is that active objects have to be connected to the event dispatcher in order to generate and receive notifications.

JEDI can be considered as an evolution of the CORBA Event Service, in which notifications are delivered not to all the receivers, as in CORBA, but only to the components that have explicitly subscribed them. A similar approach is also endorsed by commercial products like TIB/Rendezvous. For these reasons we decided to use JEDI as a case study instead of CORBA. Moreover, as we will discuss in Section 5, the usage of JEDI allows us to discuss some issues related with the refinement of architectures. In particular, since at a lower level of abstraction the JEDI event dispatcher is implemented as a set of event servers, we want to specify the attachments between active objects and these lower-level servers.

4 SPECIFYING THE CASE STUDIES IN SOME EXISTING ADLs

In this section we show how the two case studies presented in Section 3 can be specified using the ADLs ARMANI, Rapide, Darwin, Wright, and Aesop. For space reasons we focus mainly on ARMANI and Rapide, while we discuss only the most interesting features of the other ADLs. We selected ARMANI and Rapide from among the others for two main reasons: first, they endorse two opposite approaches, with ARMANI supporting the definition of topological properties and Rapide providing mechanisms for defining behaviors; second, both of them provide a relatively stable toolset for defining and analyzing architectures.

ARMANI

ARMANI [15] is an extension of ACME [6]. It is still under definition, but its basic principles seem well established, and the main functionality of its toolset appears to be robust and

[2] At instantiation time, each component communicates to its bottom bus the requests that it is able to handle.

stable. It focuses on the definition of the structural properties of an architecture (e.g., how components and connectors are attached together) and disregards the definition of behavioral properties. The ARMANI toolset provides a checker that verifies the consistency of an architectural specification.

In ARMANI a component is defined in terms of a set of *ports*. A port represents a point of interaction with the other elements of the architecture. Similarly, a connector is defined in terms of its *roles*. Each role identifies a participant in the interaction represented by the connector itself. Components and connectors can be grouped in a *system*. In the system, components and connectors form a bipartite graph, with ports and roles *attached* in a one-to-one relationship.

ARMANI supports the definition of component and connector types and provides specialization and instantiation mechanisms for these types. Also, it supports the definition of architectural constraints through the *invariant* construct. Component types, connector types, and invariants can all be collected in a *style*. A system can be defined either as an instantiation of a specific style or as an independent architecture. The ARMANI checker checks whether an architecture is compliant with the corresponding style declaration.

Definition of the C2 Style in ARMANI

The natural way of describing the C2 style in ARMANI is to define a component type representing C2 components, a connector type representing C2 buses, and a number of invariants that represent the topological constraints imposed on C2 architectures. Behavioral rules must be disregarded since they cannot be expressed by the ADL.

We define a C2 component as follows:

```
Component Type C2_compT = {
  port topPort : portTopT;
  port bottomPort : portBottomT;
  invariant size(self.ports) == 2; };
```

The two ports represent the ways C2 components interact with their top and bottom buses.[3] The invariant indicates that a C2 component can have only two ports.

C2 buses cannot be represented as ARMANI connectors, since otherwise architectures like the one in Figure 1 could not be represented (because of the connector-connector links allowed by C2). Therefore, we are forced to define a C2 bus as an ARMANI component. Also, we must define a new connector type whose instances are used to connect C2 components and C2 buses. This is due to the fact that ARMANI does not allow direct attachments between components. This results in the following two definitions:

```
Component Type C2_busT = {
invariant size(self.ports) >= 1;
invariant forall p : port in self.ports |
declaresType(p,portTopT) OR declaresType(p,portBottomT);};

Connector Type intermediary = {
role top: roleTopT; role bottom: roleBottomT;
invariant size(self.roles) == 2; };
```

[3] Port and role types are not shown for space reasons.

In the `C2_busT` definition, no ports are specified, so that architectures where buses are connected to different numbers of top and bottom components can be created. The invariant ensures that the ports declared in any instance of type `C2_busT` are either of type `portTopT` (the type of the top ports) or of type `portBottomT` (the type of bottom ports).

The ARMANI definition of C2 buses and components describes the topological constraints identified as number 1 and 2 in Section 3. The other topological constraints can be defined as global invariants. For instance, constraint number 3 can be expressed as follows:

```
invariant forall comp1: C2_compT in self.components |
  forall comp2: C2_compT in self.components |
  !connected(comp1, comp2);
```

`self.components` is the set of all the components belonging to any instantiation of the C2 style in which the invariant is defined. The predicate `connected` is provided by the ARMANI constraint language. It is true when the components it receives as parameters are attached to the same connector. The character "`!`" denotes logical negation while the character "`|`" stands for "such that".

ARMANI provides limited support for the definition of constraints whose checking requires the entire graph representing an architecture to be traversed. For instance, consider the definition of constraint number 6. It can be expressed by the following formula:

```
Invariant
 forall comp: C2_comp in self.components |
  !ReachableFromTop(comp, comp, self);
```

where `ReachableFromTop` is a C2-specific predicate we defined that accepts as parameters two components and the system that contains them. This predicate returns true if the second component can be reached from the first component, by traversing the C2 architecture in the upward direction. In ARMANI, the *design analysis* construct supports the definition of new predicates and can be used to define `ReachableFromTop` as follows:

```
Design Analysis ReachableFromTop (comp1: C2_compT,
  comp2: C2_compT, sys: C2Style): boolean =
 (Exists bus: C2_busT in sys.components |
  Exists comp: C2_compT in sys.components |
  Exists med1, med2: intermediary in sys.connectors |
  Exists port1: portBottomT in bus.ports |
  Exists port2: portTopT in bus.ports |
   attached(comp1.topPort, med1.bottom) and
   attached(bus.port1, med1.top) and
   attached(bus.port2, med2.bottom) and
   attached(med2.top, comp.bottomPort) and
   (comp==comp2 or ReachableFromTop(comp,comp2,sys)));
```

This design analysis is supposed to return true when the two components passed as parameters are attached to opposite sides (bottom and top, respectively) either of the same bus or of a chain of C2 buses and components. Unfortunately, the recursion introduced in this definition is not currently handled by the ARMANI toolset. A way to circumvent this problem is to define the implementation of

ReachableFromTop as a Java function. ARMANI, in fact, allows Java functions to be made visible in the architecture definition environment as predicates. However, this solution is quite limiting and inelegant. First, an architect is forced to use a different (programming-level) language. Second, the definition of the predicate is no longer analyzable by the ARMANI toolset.

Assuming that C2Style is the name of the style that groups all the definitions above, the architecture of Figure 1 is described by the following specification:

```
System Example: C2Style = new C2Style extended with {
 Component C1, C2, C3: C2_compT;
 Component B1: C2_busT = new C2_busT extended with {
 port top1: portTopT;
 port bottom1: portBottomT;port bottom2:portBottomT;};
 // Component B2 has a similar structure
 Connector m1, m2, m3, m4, m5: intermediary;
 Attachments {
  C1.bottomPort to m1.top; B1.top1 to m1.bottom;
  C2.topPort to m2.bottom; B1.bottom1 to m2.top;
  C2.bottomPort to m3.top; B2.top1 to m3.bottom;
  B1.bottom2 to m4.top; B2.top2 to m4.bottom;
  C3.topPort to m5.bottom; B2.bottom1 to m5.top; };}
```

In the instances of C2_busT, ports must be explicitly declared. Notice that the introduction of the fictitious intermediaries makes the Attachments part quite cumbersome.

Definition of the JEDI Style in ARMANI

The ARMANI specification of the style defined by JEDI is quite simple since it does not require the definition of complex topological constraints:

```
Style JEDI_sty = {
 role type NotifyR = {};
 //other role and port type declarations
 Connector Type JEDI_ED = {
  invariant forall r : role in self.roles |
  declaresType(r, NotifyR) OR
  //enumeration of the other legal types
  //((Un)SubscribeR, Move-inR/Move-outR) };
 Component Type ActiveObj = {
  port Notify: NotifyP;
  //enumeration of all the other legal ports };
 invariant forall comp : ActiveObj in self.components |
  forall conn : JEDI_ED in self.connectors |
   forall p : port in comp.ports |
    forall r : role in conn.roles |
    attached(r,p) -> ((declaresType(r,NotifyR) AND
                       declaresType(p,NotifyP))
                   OR //other legal types }
```

The connector type JEDI_ED defines the topological characteristics of the JEDI event dispatcher, while the component type ActiveObj represents the active objects. We have chosen to explicitly represent each kind of interaction between active objects and the event dispatcher as a pair of role and port types. The invariant ensures that each active object port is connected with a proper role (e.g., port of type SubscribeP with role of type SubscribeR, etc.).

Rapide

Rapide is an event-based ADL that has been specifically designed to support the prototyping of architectures [8]. Its toolset, in fact, supports the execution of architectural descriptions. The result of an execution is given in terms of the events that are generated and observed by each element of an architecture. These events are organized in a graph that defines the causal relationships among them.

In Rapide a component is defined by an *interface* that specifies: a) the functions and the data provided and required by the component, b) the events it is able to observe and generate, c) its behavior, and d) constraints on its behavior. An *architecture* specifies how the functions, data, and events defined for a component are connected to the corresponding functions, data, and events defined for other components; these connections are established dynamically during the execution of the architecture. Also, the architecture can define constraints on the interaction among components. The definition of styles is not explicitly supported in Rapide. However, as we will show later in this section, some simple styles can be defined as parametric architectures. A component can be associated with an implementation called a *module*. A module must *conform* to the corresponding interface. When an architecture is executed, its components wait for the occurrence of some events and react to them according to the behavior specified in the corresponding interface or, if it exists, the behavior implemented by the corresponding module.

Rapide does not provide a construct to explicitly define connectors as architectural elements. Thus, a connector has to be specified either as a connection between component constituents (functions provided and required or events generated and observed) or as an additional component (an interface). This choice depends on the complexity of the connector to be represented.

Rapide is a powerful but huge language. In our experimentation we have mostly limited ourselves to using the subset of the language that is supported by the toolset. This allowed us to actually execute and assess the specifications we defined.

Definition of the C2 Style in Rapide

As in the case of ARMANI, we define the C2 style in Rapide by specifying the properties of C2 components and buses and the way they are combined. In Rapide we can also define behavioral properties. Conversely, there is limited expressivity for the definition of topological constraints. In Rapide a C2 component is defined as an interface that is able to respond to the occurrence of two events, ReceiveFromBottom and ReceiveFromTop, and to generate two events, SendToTop and SendToBottom. C2 does not make any assumption on how components react to the occurrence of a notification or of a request. Therefore, the corresponding Rapide description does not contain any behavioral specification of such reaction.

A C2 bus is defined as follows:

```
type C2Bus is interface
action in  ReceiveFromBottom(R : Request),
           ReceiveFromTop(N : Notification);
       out SendToTop(R : Request),
```

```
        SendToBottom(N : Notification);
behavior begin
(?R: Request) ReceiveFromBottom(?R) ||> SendToTop(?R);;
(?N:Notification)ReceiveFromTop(?N) ||>SendToBottom(?N);;
end C2Bus;
```

A C2 bus observes and generates the same events handled by a C2 component. This is defined in the action section of the specification. The `in` events are the ones that the C2 bus is able to observe, while the `out` events are the ones it can generate. The behavior part of the specification indicates that a C2 bus forwards all the requests and notifications it receives from one end (top or bottom) to the opposite end, by generating a `SendToTop` or a `SendToBottom` event. Events preceding and following the symbol `||>` are, respectively, the ones generated and observed by `C2Bus` components. For simplicity, we assume that all the components attached to the top of a C2 bus observe the requests coming from its bottom.

Notice that the events in the Rapide specification do not correspond to the events (requests and notifications) that are actually generated or received by a C2 component. Instead, requests and notifications in C2 are parameters of the Rapide events.

As in ARMANI, the definition of the attachment between components is defined separately from the definitions of components. In this case, these attachments are defined by connection rules that establish the relationships between generated and observed events. The following specification describes these connection rules for the particular architecture shown in Figure 1:

```
architecture C2Example() is
 C1, C2, C3: C2Comp; B1, B2: C2Bus;
connect
  B1.SendToTop() ||> C1.ReceiveFromBottom();
  C1.SendToBottom() ||> B1.ReceiveFromTop();
  B2.SendToTop() ||> C2.ReceiveFromBottom();
  B1.SendToBottom() ||> C2.ReceiveFromTop();
  C2.SendToTop() ||> B1.ReceiveFromBottom();
  C2.SendToBottom() ||> B2.ReceiveFromTop();
  B2.SendToTop() ||> B1.ReceiveFromBottom();
  B1.SendToBottom() ||> B2.ReceiveFromTop();
  B2.SendToBottom() ||> C3.ReceiveFromTop();
  C3.SendToTop() ||> B2.ReceiveFromBottom();
end C2Example;
```

Notice from the `connect` section that events produced by a component can be notified to multiple architectural elements. For instance, since the event `SendToBottom` generated by bus `B1` is connected to `ReceiveFromTop` of both `C2` and `B2`, both `C2` and `B2` will be notified of its occurrence.

The C2 topological constraints are not easy to specify in Rapide, since the language does not provide explicit mechanisms for checking the attachments between components. Instead, it provides a *pattern language* to define expressions over the partial order of events that occur during the execution of an architecture. The pattern language can be used to specify constraints on the behavior of an architecture. These constraints implicitly impose some restrictions on the topology of the architecture itself. For instance, constraint number 5 of Section 3 could be expressed as follows:

```
never (?C: C2Comp; ?B: C2Bus; ?M: Message)
   ?C.SendToTop(?M) |> ?B.ReceiveFromTop(?M);
never (?C: C2Comp; ?B: C2Bus; ?M: Message)
   ?B.SendToTop(?M) |> ?C.ReceiveFromTop(?M);
-- equivalent expressions for the bottom sides
```

The expression with `|>` matches if the event on its left side causally precedes the one on the right side and no other event is between the two in the causal order. This operator has been defined in Rapide but not yet implemented in the toolset.

Although the constraints above are related to the topological constraint we wanted to define, they do not model the desired constraint explicitly. On the other hand, it is interesting to note that the definition of constraint number 6 of Section 3 in Rapide is substantially simpler than in the ARMANI case:

```
never (?N: Notification; ?C: C2Comp)
   ?C.SendToTop(?N) -> ?C.ReceiveFromBottom(?N);
```

The operator `->` is used to represent a causal sequence relation between events that allows other intervening, unspecified events.

Rapide does not provide an explicit construct for defining styles. The constraints presented above must be defined in an architecture specification. This means that they must be physically copied in any other architecture that defines an instance of the C2 style.

Definition of the JEDI Style in Rapide

The possibility of defining behaviors in Rapide allows the semantics of the JEDI event dispatcher to be completely reified at the architectural level:

```
type Subscription is interface
 subExpr: var SubscrExpress; subscriber: var ActiveObj;
 provides
  MMatch: function(N: Notification) return Boolean;
end Subscription;
type SubTable is array[integer] of Subscription;

type JEDI_ED (NumActiveObjs: Integer) is interface
action
 in Publish(N : Notification),
    Subscribe(S : SubscrExpress; A: ActiveObj),
    Unsubscribe(SE : SubscrExpress; A: ActiveObj),
    MoveIn(A: ActiveObj), MoveOut(A: ActiveObj);
 out Notify(N: Notification; A: ActiveObj);
behavior i: var integer := 0; ST: SubTable is
 (1..NumActiveObjs, default is new(Subscription));
begin
(?S: SubscrExpress; ?A: ActiveObj) Subscribe(?S, ?A) ||>
 i:=$i+1; ST[$i].subExpr:= ?S; ST[$i].subscriber:=?A;;
(?N: Notification) Publish(?N) ||>
 for j: integer in 1..NumActiveObjs do
  if ST[j].subscriber.Is_Nil()=False and
     ST[j].MMatch(?N)=True then
  Notify(?N, $(ST[j].subscriber)); end if; end for;;
end JEDI_ED;
```

The `JEDI_ED` interface describes the event dispatcher. The behavior part defines the reaction of the event dispatcher to the occurrence of events `Subscribe` and `Publish`. For the sake of brevity, the reactions to `MoveIn` and `MoveOut` events are not described. `ST` represents the table containing the information about the subscriptions issued by agents. Each

element of the table contains a subscription expression and a reference to the active object that has issued it. Also, the elements in the table define a function called `MMatch` that checks if notifications match the corresponding subscription expression. The implementation of this function is not shown for space reasons. When a component implementing the `JEDI_ED` interface receives a `Subscribe` event, it stores the corresponding subscription in the table `ST`. When it receives a `Publish` event, it searches the `ST` table for the agents that have issued a subscription compatible with the published notification and forwards this notification to them.

By exploiting the Rapide construct for defining parametric architectures, we can provide a general definition for the JEDI style:

```
architecture JEDI_St(NumActiveObjs: Integer)
return JEDI_Style is
 A: array [Integer] of ActiveObj is
 (1 .. NumActiveObjs, default is new(ActiveObj));
 ED: JEDI_ED(NumActiveObjs);
connect
(?A: ActiveObj; ?N: Notification)
 ?A.Publish(?N) ||> ED.Publish(?N);
(?A: ActiveObj; ?N: Notification)
 ED.Notify(?N, ?A) |||> ?A.Notify(?N);
...
end JEDI_St;
```

In the definition above, the connection rules between the events generated (received) by active objects and the ones received (generated) by the event dispatcher are specified. These rules work correctly independently of the number of active objects that are actually instantiated. This number is determined by the value of the parameter of the architecture. Notice that for C2 it is not possible to specify in Rapide a mapping between events of buses and components that is independent of the instances defining a specific architecture. This is because in C2 the topology of an architecture is not a "star" as in the JEDI case (every component connected to the same event dispatcher).

The definition of a specific architecture based on the JEDI style can be accomplished by instantiating `JEDI_Style`:

```
architecture JEDI_Inst() is
 System: JEDI_Style is JEDI_St(7);
end JEDI_Inst;
```

In this case, an architecture with seven active objects is created.

Darwin

Darwin [10] is a a *configuration language* that supports the development of architectures implemented on top of Regis. Darwin supports the definition of architectures in terms of *components*, *services*, and *bindings*. Components can be either *primitive* or *composite*. Composite components are defined in terms of their internal subcomponents. A component can *require* and *provide* services. Required and provided services are associated with each other through *bindings*. In Darwin it is not possible to define the behavior of components; these are supposed to be defined directly with Regis.

An interesting characteristic of Darwin is the possibility of defining dynamic architectures, in which it is possible to describe how components are dynamically created during the execution of the system and how they are attached to the remaining architecture. Here is an example that describes a dynamic version of a JEDI architecture:

```
component JEDI_Arch {
 provide newActiveObj;
 inst ED: JEDI_ED;
 bind newActiveObj -- dyn ActiveObj;
      ED.Subscribe -- ActiveObj.Subscribe; ... }
```

The composite component `JEDI_Arch` allows a new active object to be created each time the service `newActiveObj` is invoked. This is expressed by the first binding shown in the specification. The second binding associates the `Subscribe` service provided by the event dispatcher to the corresponding service required by active objects. The binding is expressed generically for the component type since the identity of the active objects that will be created is unknown. For the same reason, the bindings involving services provided (not required) by active objects cannot be specified in the Darwin description, since the binding rules require the identity of the components that provide a service to be known. With this restriction, the push semantics adopted by JEDI for the delivery of notifications cannot be expressed in Darwin, since JEDI requires the active objects to provide a `Notify` service to the event dispatcher.

Darwin has been used also to describe CORBA-based applications [11]. The approach is based on the idea of mapping each component identified in a Darwin architecture to a corresponding CORBA object. A tool has been implemented that supports the translation from the Darwin description of a component to CORBA IDL.

Wright

Wright has been presented in [1], with some extensions proposed in [2]. Wright shares several characteristics with ARMANI. In particular, it provides the same basic constructs (components, connectors, ports, roles, attachments). Moreover, the extensions support the definition of styles. Differently from ARMANI, Wright supports the definition of the behavior of both components and connectors. To this end, a subset of the language CSP is adopted. In CSP, behaviors are expressed algebraically in terms of patterns of events.

The following specification defines the structure of C2 buses as a connector type:

$$connector\ C2\_bus(nt : 1..,nb : 1..) =$$
$$role\ top_{1...nt} = receive \rightarrow top \,\Box\, \overline{send} \rightarrow top \,\Box\, \surd$$
$$role\ bottom_{1...nb} = receive \rightarrow bottom \,\Box\, \overline{send} \rightarrow bottom \,\Box\, \surd$$
$$glue = top.receive \rightarrow (;\ i : 1...nb \bullet bottom.\overline{send}) \rightarrow glue \,\Box$$
$$bottom.receive \rightarrow (;\ i : 1...nt \bullet top.\overline{send}) \rightarrow glue \,\Box\, \surd$$

The connector has two role types, *top* and *bottom*. The indexes indicate that the number of roles actually created depends on the parameters of the connector. For both role types the events they can receive/produce are specified. The glue

describes the behavior of the connector. In the example, the glue specifies that whenever the bus receives a request from its bottom or a notification from its top, it forwards this request or notification to all the roles on the opposite side.

Thanks to its capability of describing behaviors and to the recent introduction of constructs for supporting the definition of styles, Wright seems to be suitable for our purposes (except for its connector semantics, as discussed in Section 5). However, since it currently does not provide a toolset to support the definition and analysis of styles and architectures, we could not assess its features in greater detail.

Aesop

Aesop [4] is not properly an ADL. It is an environment for defining style-dependent architecture definition environments. It allows a programmer to define a style in terms of component, connector and connection rule types. In general, these building blocks are quite similar to the ones provided by ARMANI and Wright. However, Aesop does not provide an architectural language for defining styles. Instead, a developer of styles must work at the programming language level to customize the environment on the basis of the rules defined by the style. The programming environment that is used is an extension of Tcl/Tk. Aesop also provides a tool integration environment that makes it possible to integrate analysis tools developed for specific styles. We argue that the main limitation of Aesop is that styles are not defined in an ADL. This means that it is difficult to reuse and specialize styles, or to communicate them to architects.

5 EVALUATION OF THE EXPERIENCE AND NEW REQUIREMENTS FOR ADLs

Our experimentation started with the goal of identifying the proper set of ADLs to support the definition of libraries of reusable middleware-induced styles. In this experience, we have realized that while many of our needs are addressed by different ADLs, no ADL fully addresses all of them. In this section we focus on the evaluation of our experience and on the identification of the characteristics that an ADL should have in order to be suitable for our needs.

Style Definition

ADLs should be able to define styles, that is, a coherent collection of component types, connector types and stylistic constraints. Moreover, they should provide a mechanism for exploiting a style in the definition of an architecture. ARMANI, the extended version of Wright, and Aesop explicitly support the definition of styles. In ARMANI, an architecture can extend a style by defining new architecture-specific constraints. The ARMANI constraint checker is able to check if the constraints defined in a style are satisfied by instantiated architectures. Conversely, Rapide and Darwin provide quite limited support for the definition of styles. Both languages provide a construct to define parametric architectures that, as we have shown in Section 4, can be used to describe styles in limited cases. Even in these cases, this solution introduces some restrictions. For instance, the definition of the JEDI style in Rapide (see Section 4) does not enable the corresponding architectures to be composed of different coexisting specializations of type `ActiveObj`. In fact, active objects are instantiated directly in the style definition which is only aware of the definition of type `ActiveObj`. The only way to specialize the generic definition of this type is to overwrite it.

It is also useful to specialize styles as substyles. The definition of substyles, on the one side, enables the reuse of existing architectural descriptions and, on the other side, provides an organization of styles that can guide the architect in selecting the proper paradigm for a particular application domain. For instance, it is intuitive that most event-based middlewares specialize a general event-based style, in which the components and connectors interact through the basic publish, subscribe, and notify operations. Each specialization can introduce new operations (for instance, JEDI adds mechanisms to support the temporary disconnection of a component) or can redefine the semantics of existing operations (for instance, each middleware defines its own specific semantics for matching notifications with subscriptions). ARMANI fully supports the definition of substyles. In Rapide, defining substyles could be accommodated by subtyping the components of the superstyle. However, while some subtyping conditions between interface definitions are defined, the behavioral part of interfaces cannot be inherited (see below for more on behavior). Moreover, the connection rules defined in a Rapide architecture for a supertype do not seem to apply to any specialization of this supertype.

Topological Constraints

In a style definition, we do not want to specify the attachments or connections of specific components. Instead, we would like to provide some general topological constraints that must be respected by any specific instantiation of the component and connector types defined in the style. In ARMANI and the extended Wright, invariants support the definition of such constraints in a powerful way. As we have mentioned in Section 4, a limitation of ARMANI is the fact that it does not currently support the definition of constraints that contain recursive rules. In Rapide it is possible to define constraints on the sequence in which events are generated and/or received by components. We have shown that this kind of constraint can be used to establish certain restrictions on the structure of the architecture, albeit implicitly.

Behaviors

Topological constraints are not enough to define styles. The description of the behavior of components and connectors is at least equally important. If this description is not available, architects cannot have a clear idea of how the elements of the style (and in our case the corresponding implementations in the middleware) work and how they can be used. ARMANI, Aesop, and Darwin do not address this aspect. Rapide provides powerful linguistic constructs for specifying behaviors.

Still, it has some limitations when a component is specialized or when it is implemented in terms of a module. In both these cases, the behavior of the original interface definition is not guaranteed to be preserved. Compatibility checks are performed only as far as the static part of the definitions is concerned. An approach to checking the behavioral *conformance* between supertypes and subtypes is discussed in [14].

Connectors

The importance of explicitly describing the connectors of an architecture has been stated in several seminal papers on software architectures (see for instance [20, 7]). Surprisingly, the languages that provide an explicit construct for defining connectors associate with them a semantics that is too restrictive. For instance, in ARMANI, each connector role can be attached to only one component port. This means that to create a multicast connector, we need to define a number of roles equal to the number of components to be attached to that connector. This approach is more cumbersome than the one adopted in Rapide, in which generated events can be connected to more than one observed event. Another important limitation of explicit connector languages discussed in Section 4 is that connectors (and also components) cannot be attached together.

This model for connectors seems influenced by the characteristics of sockets, pipes, and (remote) procedure calls. More modern kinds of connectors, such as event dispatchers, ORBs (Object Request Brokers), and multicast channels, highlight the limitation of this semantics. It could be argued that these connectors can be described as components. However, we believe that architectural definitions are more readable and clear when the special purpose of these architectural elements for component interoperability is made explicit. Also, as we have discussed in Section 4, where we present the definition of the C2 style in ARMANI, the definition of these connectors as components adds a new level of indirection. In fact, we needed to define an intermediate, artificial connector type to attach the "real" connectors to the actual components of a C2 architecture.

Refinement of Components and Connectors

An important requirement for both connectors and components is the possibility of refining their internal structure in terms of the composition of other components and connectors. Such refinement supports the co-existence of different levels of abstraction in an architecture. In general, for a refinement to be valid, all the elements that belong to the public interface of a component/connector must be offered by some component/connector defined in any refinement.

As an example of refinement, consider the case of the JEDI event dispatcher. So far, we have defined the event dispatcher as a simple connector. However, its implementation is distributed. In particular, it is composed of a hierarchy of *event servers*. At the architectural level, the internal structure of the event dispatcher can be represented by a refinement in which the operations defined in the event servers are replicas of the operations belonging to the interface of the event dispatcher. Having defined this refinement, the attachments between active objects and the event dispatcher should be refined as well, by specifying to which event server each active object is attached. This refinement of attachments, in fact, would be extremely useful for evaluating the performance of an architecture as a function of the distribution of the active objects over the hierarchy of servers.

While all the languages we considered, except ARMANI, support the refinement of components and connectors, none of them supports the refinement of the corresponding attachments. We argue instead that, in conjunction with the refinement of a component (connector), it should be possible to refine the attachments of this component (connector) with the other architectural elements to which it is attached. This refinement should be *conservative* in the sense that the refined attachments should preserve the characteristics of the original ones (e.g., if a port `A` is attached to a role `B` in the original architecture, then, in the detailed architecture, `A` should be attached to some element of the refinement of `B`). Powerful linguistic constructs to express generic mappings between the elements of an architecture and their refinements are provided by SADL [16]. The limitation of such approach is that it does not provide explicit guidelines on how to perform this refinement.

6 CONCLUSION

The general lesson we learned from our experience is that the top-down approach adopted by the software architecture community in the development of languages and tools seems in many ways to ignore the results that practitioners have achieved (in a bottom up way) in the definition of middlewares. Middlewares have demonstrated their usefulness and effectiveness in a number of practical cases. The software architecture community has now the potential to formalize these achievements in expressive and usable ADLs and, more generally, to coordinate the definition of support technology for the development of middleware-based applications.

Our next step is to continue our exploration of linguistic mechanisms and modeling techniques that allow architecture models to capture middleware-induced architectural styles. We are broadening our search space by looking at UML and, in general, languages that are not strictly considered ADLs. Our longer term objective is to implement an environment that supports the definition of architectures by providing a library of styles induced by specific middlewares. Given an architecture defined according to such a style, this environment will be able to partially automate the implementation of the architecture on the corresponding middleware.

ACKNOWLEDGEMENTS

We wish to thank Alfonso Fuggetta and Debra Richardson who reviewed this paper and gave us useful suggestions on

its structure and content. Also, we thank Neno Medvidovic, Peyman Oreizy, Arthur Reyes, and Alex Wolf who contributed to the clarification of the issues we presented. Finally, we thank Bob Monroe who assisted us with ARMANI.

Elisabetta Di Nitto worked on this paper when she was visiting University of California, Irvine. This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under grant number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, 1997.

[3] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploting an Event-Based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, Kyoto (Japan), April 1998.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, December 1994.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995.

[6] D. Garlan, R. T. Monroe, and D. Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, November 1997.

[7] D. Garlan and M. Shaw. An Introduction to Software Architectures. *Advances in Software Engineering and Knowledge Engineering*, 1993.

[8] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[9] J. Magee, N. Dulay, and J. Kramer. Regis: A Constructive Development Environment for Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5), September 1994.

[10] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of SIGSOFT '96 Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.

[11] J. Magee, A. Tseng, and J. Kramer. Composing Distributed Objects in CORBA. In *Proceedings of ISADS'97*, Berlin, Germany, April 1997.

[12] N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

[13] N. Medvidovic, D. Rosenblum, and R. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles (CA), May 1999.

[14] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Type Theory for Software Architectures. Technical Report UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.

[15] R. Monroe. ARMANI Language Reference Manual. CMU Technical Report in preparation.

[16] M. Moriconi, X. Qian, and R. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[17] Object Management Group. CORBA Services: Common Object Services Specification. Technical report, OMG, July 1997.

[18] P. Oreizy, N. Medvidovic, R. Taylor, and D. Rosenblum. Software Architecture and Component Technologies: Bridging the Gap. In *Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures*. Monterey, CA, January 1998.

[19] OVUM. OVUM Evaluates Middleware. Technical report, OVUM Ltd., 1996.

[20] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[21] M. Shaw and P. Clements. Toward Boxology: Preliminary Classification of Architectural Styles. In *Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Francisco (CA) USA, October 1996*, San Francisco (CA) USA, October 1996.

[22] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[23] K. Sullivan, J. Socha, and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, 1997.

[24] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6), June 1996.

A Type Theory for Software Architectures

Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor

Technical Report UCI-ICS-98-14

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
{neno,dsr,taylor}@ics.uci.edu

April 1998

ABSTRACT

Software architectures have the potential to substantially improve the development and evolution of large, complex, multi-lingual, multi-platform, long-running systems. However, in order to achieve this potential, specific architecture-based modeling, analysis, and evolution techniques must be provided. This paper motivates and presents one such technique: a type theory for software architectures, which allows flexible, controlled evolution of software components in a manner that preserves the desired architectural relationships and properties. Critical to the type theory is a taxonomy that divides the space of subtyping relationships into a small set of well defined categories. The paper also investigates the effects of large-scale development and off-the-shelf reuse on establishing type conformance between interoperating components in an architecture. An existing architecture is used as an example to illustrate a number of different applications of the type theory to architectural modeling and evolution.[1]

1. INTRODUCTION

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are commonly estimated to be as high as 60% of overall development costs [7]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus supporting only limited reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

An explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key property is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability: few specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [16]. Each of them may evolve. Our work to date has addressed the evolution of connectors and topologies [14, 15, 19, 25]. This paper proposes a technique for evolving software components. This technique has resulted from the recognition that researchers in software architectures, and particularly in architecture description languages (ADLs), can learn from extensive experience in the area of PLs, and object-oriented languages (OOPLs) in particular. The particular lesson in this case is that an existing software module can evolve in a controlled manner via subtyping.

Our approach to component evolution is indeed based on a type theory. We treat each component specification in an architecture as a type and support its evolution via subtyping. However, while PLs (and several existing ADLs [5, 6, 10]) support a single subtyping mechanism, we have demonstrated that architectures may require multiple subtyping mechanisms, many of which are not commonly supported in PLs [13]. Therefore, existing PL type theories are inadequate for use in software architectures.

Beyond evolution, types are also useful in establishing certain correctness criteria about a program or an architecture. Several existing ADLs support type checking (e.g., Aesop [5], Darwin [12], Rapide [10], and UniCon [23]). However, as with most all of the existing PLs, these ADLs essentially establish simple syntactic matches among interacting components. Our approach also establishes semantic conformance of components.

Furthermore, all existing type checking mechanisms regard types as either compatible or incompatible. Although it is beneficial to characterize component compatibility in this way, determining the *degree* of compatibility, and thus the potential for component interoperability, is more useful. One of the goals of the software architecture and component-based-development communities is to provide more extensive support for building

1. Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-97-2-0021 and F30602-94-C-0218, and by the Air Force Office of Scientific Research under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973.

systems out of existing parts. Those parts will typically not perfectly conform to each other. We have demonstrated that partially mismatched components can in certain cases still be effectively combined in an architecture [14, 15]. Establishing the degree of compatibility can also help determine the amount of work necessary to retrofit a component for use in a system.

The contributions of this paper are threefold:

- a taxonomy that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces;
- a flexible type theory for software architectures that is domain-, style-, and ADL-independent. By adopting a richer notion of typing, this theory is applicable to a broad class of design and reuse circumstances; and
- an approach to establishing type conformance between interoperating components in an architecture. This approach is better suited to support the "large scale development with off-the-shelf reuse" philosophy on which architecture research is largely based than other existing techniques.

The remainder of this paper is organized as follows. Section 2 briefly discusses the architecture that is a basis of examples used throughout the paper to illustrate the concepts of the type theory. Section 3 introduces and discusses the general principles of the architectural type theory. Section 4 formally defines a particular instance of the type theory, or a type system, that exhibits the necessary properties for component evolution and architectural type checking. A discussion of our results to date, conclusions, and future work round out the paper.

2. EXAMPLE ARCHITECTURE

To illustrate the concepts in this paper, we use the architecture shown in Fig. 1. This architecture resulted from the case study conducted during the Second International Software Architecture Workshop (ISAW-2) [26]. The system that the architecture models is a *call center customer care* (C4) system for a large telecommunications company. The architecture includes several subsystems identified by the telephone company:

- Corporate databases and billing — customer account management;
- Network Operations Support System (NOSS) — management and provisioning of the physical network;
- Downstream systems — e.g., long-distance carrier services, 911 service, voice mail, and so forth;
- "Quick" service — enables customers to directly communicate with the system; and
- "C4 core" — manages the above parts of the system and provides support for service negotiations, account management, and trouble-call management.

The architecture in Fig. 1 addresses most of the major system requirements and has previously been discussed in more detail [27]. It is modeled in the C2 architectural style: a component communicates with components above and below it in the architecture by sending asynchronous messages, which are then routed to the appropriate components by connectors [25]. Although some concerns have been voiced about the suitability of the C2 style for this particular application, the specifics of the style and of the architecture are not critical for the purpose of demonstrating the concepts introduced in this paper. Instead, we chose this particular example because it has well-defined requirements from an actual project and describes a large-scale problem, for which software architectures are particularly well suited.

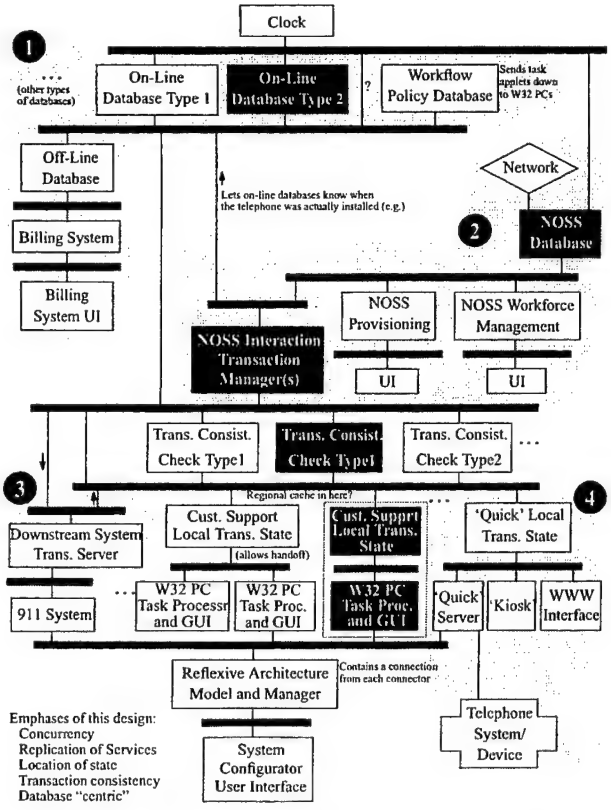


Fig. 1. Call Center Customer Care (C4) System architecture:
(1) Corporate databases and billing;
(2) Network Operations Support System (NOSS);
(3) Downstream systems;
(4) "Quick" service.
Highlighted components have been modeled extensively.

We have extensively modeled the components highlighted in Fig. 1. These components were selected because they constitute a logical subsystem (basic customer service request handling) and exhibit interesting properties. The examples used in the remainder of the paper will be drawn from this modeling effort.

3. GENERAL PRINCIPLES OF THE TYPE THEORY

Explicit treatment of types enables *subtyping*, the evolution of a given type to satisfy new requirements, and *type checking*, the determination of whether instances of one type may be legally used in places where another type is expected. This notion of legality can help software developers keep program semantics close to programmer intentions, and thus discipline the evolution and (re)use of objects. Furthermore, a combination of type declarations and type checking supports source code understandability and, ultimately, the generation of efficient executable code.

A useful overview of PL subtyping relationships is given by Palsberg and Schwartzbach [21]. They describe a consensus in the OO typing community regarding the definition of a range of OO typing relationships. *Arbitrary subclassing* allows any class to be declared a subtype of another, regardless of whether they share a common set of methods. *Name compatibility* demands that there exist a shared set of method names available

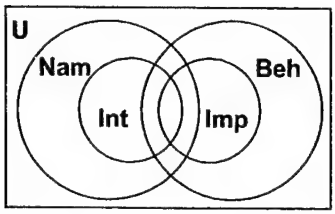


Fig. 2. A framework for understanding OO subtyping relationships as regions in a space of type systems.

in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass must preserve the interface of the superclass, while possibly extending it. *Behavioral conformance* [2, 3, 11, 29] allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. Finally, *strictly monotone subclassing* additionally demands that the subtype preserve the particular implementations used by the supertype.

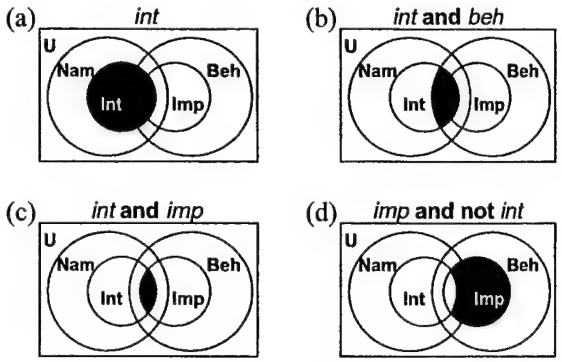
Protocol conformance goes beyond the behavior of individual methods to specify constraints on the order in which methods may be invoked. Explicitly modeling protocols has been shown to have practical benefits [1, 9, 18, 28, 29]. However, component invariants and method preconditions and postconditions can be used to describe all state-based protocol constraints and transitions. Thus, our notion of behavioral conformance implies protocol conformance, and we do not address them separately.

We have developed a framework for understanding these subtyping relationships as regions in a space of type systems, shown in Fig. 2. The entire space of type systems is labeled *U*. The regions labeled *Int* and *Beh* contain systems that demand that two conforming types share interface and behavior, respectively. The *Imp* region contains systems that demand that a type share particular implementations of all supertype methods, which also implies that types preserve the behavior of their supertypes. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance.

Each subtyping relationship described in [21] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the Int and Beh regions and is expressed as *int* **and** *beh* (Fig. 3b). Each region in Fig. 2 encompasses a set of variations of a given subtyping relationship, rather than a single relationship. Thus, for example, the different flavors of the behavioral conformance relationship, described by Zaremski and Wing [29], represent different points in the *int* **and** *beh* subspace. The architectural type system we propose in the next section also represents a selection of individual points within the different subspaces.

This type theory was motivated by our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [13, 15]. Several examples are shown in Fig. 3:

- *interface conformance* (Fig. 3a) has proven useful in interchanging components that communicate via asynchronous message passing (e.g., C2 architectural style [25]), without affecting dependent components in a given architecture;
- *behavioral conformance* (Fig. 3b) guarantees correctness during component substitution;
- *strictly monotone subclassing* (Fig. 3c) enables extension of the behavior of an existing component while preserving correctness relative to the rest of the architecture;
- *implementation conformance with different interfaces* (Fig. 3d) allows a component to be fitted into an alternate domain of discourse (e.g., by using software adaptors [28]);
- *multiple conformance mechanisms* allow creation of a new type by subtyping from several types using different subtyping mechanisms.



ig. 3. Examples of component subtyping relationships we have encountered in practice.

Note that we referred to the first three examples (Fig. 3a-c) using the terminology from the Palsberg-Schwartzbach taxonomy. However, while in OOPLs the three subtyping mechanisms would be provided by three separate languages, in architectures they all need to be supported by the same ADL and may actually be applied to components in a single architecture. Also, the example in Fig. 3d does not have a corresponding OOPL mechanism, further motivating the need for a flexible type theory for software architectures.

At the same time, by giving a software architect more latitude in choosing the direction in which to evolve a component, we allow some potentially undesirable side effects. For example, by preserving a component's interface, but not its behavior, the component and its resulting subtype may not be interchangeable in a given architecture. However, it is up to the architect to decide whether to preserve architectural type correctness, in a manner similar to America [2], Liskov and Wing [11], Leavens et al. [3], and others (depicted in Fig. 3b), or simply to enlarge the palette of design elements in a controlled manner, in order to use them in the future.

4. ARCHITECTURAL TYPE SYSTEM

In [13] we discussed the types of syntactic constructs needed in an ADL in order to support our type theory. In this section we present a type system for software architectures that instantiates the type theory. The two possible applications of an architectural type theory—evolution of existing components by software architects, and type checking of architectural descriptions—are discussed below in Sections 4.2 and 4.3, respectively. All definitions are specified in Z, a language for modeling mathematical objects based on first order logic and set theory [24]. Z uses standard logical connectives ($\vee$, $\wedge$, $\Rightarrow$,

etc.) and set-theoretic operations ($\mathbb{P}$ to denote sets, $\in$, $\cup$, , etc.).

4.1. Components

Every component specification is an *architectural type*. We distinguish architectural types from *basic types* (e.g., integers, strings, arrays, records, etc.). Unlike OOPLs, in which objects communicate by passing around other objects, in software architectures components are distinguished from the data they exchange during communication. In other words, a "component" in the sense in which we use it here is never passed from one component in an architecture to another.

A component has a name, a set of interface elements, an associated behavior, and (possibly) an implementation. Each interface element has a direction indicator (*pro*vided or *req*uired), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type.

A component's behavior consists of an invariant and a set of operations. The invariant is used to specify any protocol constraints on the use of the component. Each operation has preconditions, postconditions, and (possibly) a result. Since operations are decoupled from interface elements, they also provide a set of variables used to express preconditions and postconditions. Like interface elements, operations can be *pro*vided or *req*uired. Only provided operations will have an implementation in a given component. The preconditions and postconditions of required operations express the *expected* semantics for those operations. Formal specification of an architectural type (component) is shown in Fig. 4.[2]

In the interest of space, Fig. 4 does not specify the relationship of component invariants to operation pre- and postconditions. This relationship can be summarized semi-formally as follows. Given a component C and operation O provided by C, for all valid input states of O that satisfy C's invariant and O's precondition, there exists a valid output state that satisfies both O's postcondition and C's invariant.

Since we separate the interface from the behavior, we define a function, *int_op_map*, which maps every interface element to an operation of the behavior. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation's result type. This property directly enables a single operation to export multiple interfaces.

An example of component specification, given in C2's ADL, is shown in Fig. 5. Only partial specifications of the *On-Line Database* (*OnLineDB*) and *NOSS Interaction Transaction Manager* (*NOSS_Mgr*) components from Fig. 1 are shown due to space constraints. For example, we show only the portion of *NOSS_Mgr* that is intended to interact with *OnLineDB*.

In a C2 architecture, a component has no dependencies on components below it (principle of *substrate independence*). Thus, *OnLineDB* has no dependencies on *NOSS_Mgr* and consequently has no required services. An example of mapping two interface elements to the same operation is given in the *NOSS_Mgr* component: both *ActivatePhoneLine* and *ActivateSpecialLine* (e.g., information or emergency) can be mapped to *op2* if *SPEC_NUM* is a subtype of *PHONE_NUM*.

4.2. Architectural Type Conformance

Informally, a subtyping relation, $\leq$, between two components, C_1 and C_2, is defined as the disjunction of the *nam*, *int*, *beh*, and *imp* relations shown in Fig. 2:

$$(\forall C_1, C_2 : \mathtt{Component})\ (C_2 \leq C_1 \Leftrightarrow C_2 \leq_{\mathtt{nam}} C_1 \vee C_2 \leq_{\mathtt{int}} C_1 \vee C_2 \leq_{\mathtt{beh}} C_1 \vee C_2 \leq_{\mathtt{imp}} C_1)$$

We consider these four relations in more detail below.

4.2.1. Name Conformance

Name conformance requires that a subtype share its supertype's interface element names and all interface parameter names. The subtype may introduce additional interface elements and additional parameters to existing interface elements. Two interface elements in a single component can have identical names, but then their sets of parameter names must differ. Name conformance rules are formally specified in Fig. 6.

Note that the possibility of introducing additional parameters to existing interface elements is different from method overloading and is typically not allowed in a PL. However, software architectures are at a level of abstraction

Variable

$name : STRING$
$type : BASIC\_TYPE$

Int_Element

$dir : DIRECTION$
$name : STRING$
$params : \mathbb{P}\ Variable$
$result : BASIC\_TYPE$

Operation

$vars : \mathbb{P}\ Variable$
$precond : Logic\_Pred$
$postcond : Logic\_Pred$
$result : BASIC\_TYPE$
$dir : DIRECTION$
$implementation : \mathrm{seq}\ STATEMENT$

$dir = req \Rightarrow implementation = \varnothing$

Component

$Basic\_Type\_Conformance$
$name : STRING$
$interface : \mathbb{P}\ Int\_Element$
$invariant : Logic\_Pred$
$operations : \mathbb{P}\ Operation$
$int\_op\_map : Int\_Element \rightarrow Operation$

$\mathrm{dom}\ int\_op\_map = interface$
$\mathrm{ran}\ int\_op\_map = operations$

$\forall ie : Int\_Element;\ o : Operation \mid$
$ie \in interface \wedge o \in operations \bullet$
$(ie, o) \in int\_op\_map$
$\Leftrightarrow$
$ie.dir = o.dir \wedge$
$(ie.result, o.result) \in Basic\_Conf \wedge$
$(\forall iv : Variable \mid iv \in ie.params \bullet$
$\exists ov : Variable \mid ov \in o.vars \bullet$
$(ov.type, iv.type) \in Basic\_Conf)$

Fig. 4. Z specification of architectural types (components). Relation *Basic_Conf* is defined in the schema *Basic_Type_Conformance* and relates two basic types, the first of which is a supertype of the second.

2. Capitalized identifiers are the basic (unelaborated) types in a Z specification. Trivial schemas and schemas whose meanings are obvious are omitted for brevity.

```
Component OnLineDB is
  State
    Customers  : set CUST;
    CustIds    : set CUST_ID;
    CustById   : CUST_ID  →  CUST;
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
  Invariant
    #Customers ≥ 0;
  Operations
    prov op1:
      Let   c:CUST;
      Pre   c ∉ Customers;
      Post  Customers' = Customers ∪ {c};
    prov op2:
      Let   c:CUST;
      Pre   c ∈ Customers;
      Post  Customers' = Customers - {c};
    prov op3:
      Let   id:CUST_ID;
            c:CUST;
      Pre   id ∈ CustIds ∧ c ∉ Customers;
      Post  c ∈ Customers ∧ CustById(id) = c;
    prov op4:
      Let   id:CUST_ID;
      Pre   id ∈ CustIds;
      Post  result = CustById(id);
  Map
    ip1 -> op1(new_cust -> c);
    ip2 -> op2(cust -> c);
    ip3 -> op3(cust -> id, new_rec -> c);
    ip4 -> op4(cust -> id);
end OnLineDB;

Component NOSS_Mgr is
  State
    Numbers : ADDR  →  PH_NUM;
  Interface
    prov ip1:  ActivatePhoneLine(a:ADDR; num:PH_NUM);
    prov ip2:  ActivateSpecialLine(a:ADDR; n:SPEC_NUM);
    prov ip3:  DeactivatePhoneLine(a:ADDR; num:PH_NUM);
    req ir1:   AddNewCust(new_cust:CUST);
    req ir2:   RemoveCust(cust:CUST_ID);
  Operations
    prov op1:
      Let   addr:ADDR;
            pn:PH_NUM;
      Pre   pn ∉ Numbers(addr);
      Post  pn ∈ Numbers'(addr);
    prov op2:
      Let   addr:ADDR;
            pn:PH_NUM;
      Pre   pn ∈ Numbers(addr);
      Post  pn ∉ Numbers'(addr);
    req or1:
      Let   c:CUST;
            cust_sv:STATE_VARIABLE;
      Pre   c ∉ cust_sv;
      Post  c ∈ cust_sv';
    req or2:
      Let   c:CUST;
            cust_sv:STATE_VARIABLE;
      Pre   c ∈ cust_sv;
      Post  c ∉ cust_sv';
  Map
    ip1 -> op1 (a -> addr, num -> pn);
    ip2 -> op1 (a -> addr, n -> pn);
    ip3 -> op2 (a -> addr, num -> pn);
    ir1 -> or1 (new_cust -> c);
    ir2 -> or2 (cust -> c);
end NOSS_Mgr;
```

Fig. 5. Partial specifications of the *OnLineDB* and *NOSS_Mgr* components from Fig. 1 in the C2 ADL. Basic type STATE_VARIABLE is discussed in Section 4.3.1. Labels for interface elements and operations (e.g., ip1, or2) are a notational convenience.

that is above source code and this feature may be supported by the architecture implementation infrastructure. For example, the implementation of the C2 class framework [14] allows the sender of the communication message to include parameters the receiver component does not expect; those parameters are simply ignored by the receiver. It is up to the architect to decide whether such a situation should be permitted in a given architecture.

Param_Name_Conformance

$$Prm\_Nam\_Conf : Int\_Element \leftrightarrow Int\_Element$$

$$\forall ie1, ie2 : Int\_Element \bullet (ie1, ie2) \in Prm\_Nam\_Conf \Leftrightarrow (\forall p1 : Variable \mid p1 \in ie1.params \bullet \exists p2 : Variable \mid p2 \in ie2.params \bullet p1.name = p2.name \wedge (\forall p3, p4 : Variable \mid p3 \in ie1.params \wedge p4 \in ie2.params \bullet p3.name = p4.name \Leftrightarrow (p1 = p3 \wedge p2 = p4) \vee (p1 \neq p3 \wedge p2 \neq p4)))$$

Name_Conformance

$$Param\_Name\_Conformance$$

$$Nam\_Conf : Component \leftrightarrow Component$$

$$\forall c1, c2 : Component \bullet (c1, c2) \in Nam\_Conf \Leftrightarrow (\forall ie1 : Int\_Element \mid ie1 \in c1.interface \bullet \exists ie2 : Int\_Element \mid ie2 \in c2.interface \bullet ie1.name = ie2.name \wedge (ie1, ie2) \in Prm\_Nam\_Conf)$$

Fig. 6. Name conformance.

4.2.2. Interface Conformance

Name conformance is a rather weak conformance requirement and we have encountered it in practice only as part of the stronger *interface* conformance relationship. Component C_2 is an interface subtype of C_1 if and only if it provides at least (but not necessarily only) the interface elements provided by C_1 with identical direction indicators, and *matching* parameters and results for each interface element. Two parameters belonging to the two components' interface elements match if and only if they have identical names (*Param_Name_Conformance* schema in Fig. 6) and each

Param_Conformance

$$Basic\_Type\_Conformance$$

$$Param\_Name\_Conformance$$

$$Prm\_Conf : Int\_Element \leftrightarrow Int\_Element$$

$$\forall ie1, ie2 : Int\_Element \bullet (ie1, ie2) \in Prm\_Conf \Leftrightarrow (ie1, ie2) \in Prm\_Nam\_Conf \wedge (\forall p1, p2 : Variable \mid p1 \in ie1.params \wedge p2 \in ie2.params \wedge p1.name = p2.name \bullet (p2.type, p1.type) \in Basic\_Conf)$$

Interface_Conformance

$$Name\_Conformance$$

$$Param\_Conformance$$

$$Int\_Conf : Component \leftrightarrow Component$$

$$\forall c1, c2 : Component \bullet (c1, c2) \in Int\_Conf \Leftrightarrow (c1, c2) \in Nam\_Conf \wedge (\forall ie1, ie2 : Int\_Element \mid ie1 \in c1.interface \wedge ie2 \in c2.interface \bullet ie1.dir = ie2.dir \wedge (ie1, ie2) \in Prm\_Nam\_Conf \Rightarrow (ie1, ie2) \in Prm\_Conf \wedge (ie1.result, ie2.result) \in Basic\_Conf)$$

Fig. 7. Interface conformance.

```
Component CallPlanDB is subtype OnLineDB(int)
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
    prov ip5:  EnterCallPlan(c:CUST_ID; p:CALL_PLAN);
  Invariant
    . . .
  Operations
    prov op10:
      Let   id:CUST_ID;
            plan:CALL_PLAN;
      Pre   id ∈ CustIds ∧ plan ∈ CallPlans;
      Post  plan ∈ CustPlans(CustById(id));
    . . .
  Map
    ip5 -> op10(c -> id, p -> plan);
    . . .
end CallPlanDB;
```

Fig. 8. *CallPlanDB* is an interface subtype of *OnLineDB*.

parameter type of C_1 is a subtype of the corresponding parameter type of C_2 (*contravariance of parameters*, defined in the *Param_Conformance* schema in Fig. 7). The results of two corresponding interface elements match if the result type in C_1 is a supertype of the result type in C_2 (*covariance of result*). For each interface element, the subtype must provide at least (but not necessarily only) the parameters that match the supertype's parameters. Interface conformance rules are formally specified in Fig. 7.

For example, component *CallPlanDB*, shown in Fig. 8, is an interface subtype of *OnLineDB* from Fig. 5. It does not matter what the invariant, operations, and *int_op_map* of *CallPlanDB* are for this relationship to hold. These details have thus been omitted.

4.2.3. Behavior Conformance

Behavior conformance requires that the invariant of the supertype be ensured by that of the subtype. Furthermore, each operation of the supertype must have a corresponding operation in the subtype (the subtype can also introduce additional operations), where the subtype's operation has the same direction indicator as the supertype's, the same or weaker preconditions, same or stronger postconditions, and preserves result covariance.

No constraints are placed on the relationship between the types of the supertype's and subtype's corresponding operation variables. This relationship can vary, but is always an instance of one of the two cases depicted in Fig. 9. Thus, any

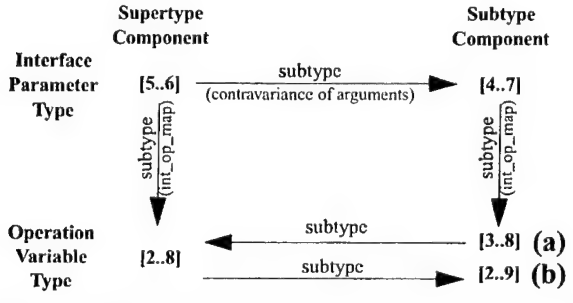


Fig. 9. Contravariance of arguments and the *int_op_map* function do not guarantee a particular relationship between supertype's and subtype's operation variable types (illustrated using integer subranges): (a) supertype component's variable type is a supertype of subtype component's; (b) supertype component's variable type is a subtype of subtype component's.

Oper_Conformance

Basic_Type_Conformance
Logical_Implication
$Oper\_Conf : Operation \leftrightarrow Operation$

$$\begin{aligned}
&\forall o1, o2 : Operation \bullet \\
&\quad (o1, o2) \in Oper_Conf \\
&\qquad \Leftrightarrow \\
&\quad (\forall v1 : Variable \mid v1 \in o1.vars \bullet \\
&\quad \exists v2 : Variable \mid v2 \in o2.vars \bullet \\
&\qquad (v1.type, v2.type) \in Basic_Conf \vee \\
&\qquad (v2.type, v1.type) \in Basic_Conf) \wedge \\
&\quad (o1.precond, o2.precond) \in Logic_Impl \wedge \\
&\quad (o2.postcond, o1.postcond) \in Logic_Impl \wedge \\
&\quad (o1.result, o2.result) \in Basic_Conf
\end{aligned}$$

Behavior_Conformance

Oper_Conformance
Logical_Implication
$Beh\_Conf : Component \leftrightarrow Component$

$$\begin{aligned}
&\forall c1, c2 : Component \bullet \\
&\quad (c1, c2) \in Beh_Conf \\
&\qquad \Leftrightarrow \\
&\quad (c2.invariant, c1.invariant) \in Logic_Impl \wedge \\
&\quad (\forall o1 : Operation \mid o1 \in c1.operations \bullet \\
&\quad \exists o2 : Operation \mid o2 \in c2.operations \bullet \\
&\qquad o1.dir = o2.dir \wedge \\
&\qquad (o1, o2) \in Oper_Conf)
\end{aligned}$$

Fig. 10. Behavior conformance. *Logic_Impl* is a relation that denotes that the first element in the relation implies the second.

relationship between the variable types is allowed so long as the proper relationships between operation pre- and postconditions are maintained. The rules for behavior conformance are specified in Fig. 10.

Fig. 11 shows an example of behavior conformance. Component *BoundedDB* is a behavior subtype of *OnLineDB* from Fig. 5. The interface and *int_op_map* of *BoundedDB* are unimportant for this relationship to hold. These details have thus been omitted. *BoundedDB* is a behavior subtype of *OnLineDB* because it provides (at least) the same operations as *OnLineDB* and its invariant (number of customers, representing the size of the database), is between zero and one million, inclusive, which implies *OnLineDB*'s invariant (number of customers is zero or greater).

The subtyping relationship that results from the combination of the *Behavior_Conformance* and *Interface_Conformance* schemas (in particular, the *Beh_Conf* and *Int_Conf* relations they define), and the mapping function, *int_op_map*, represents a point in the region depicted in Fig. 3b. This relationship is similar to other notions of behavioral subtyping [2, 3, 11] in that it guarantees substitutability between a supertype and a subtype in an architecture.

```
Component BoundedDB is subtype OnLineDB(beh)
  Interface
    . . .
  Invariant
    0 ≤ #Customers ≤ 1000000;
  Operations
    [OnLineDB operations]
    . . .
  Map
    . . .
end BoundedDB;
```

Fig. 11. *BoundedDB* is a behavior subtype of *OnLineDB*.

```
Component OnLineDB-2 is subtype OnLineDB(int and beh)
  State
    Customers  : set CUST;
    CustIds    : set CUST_ID;
    CustById   : CUST_ID  →  CUST;
    CallPlans  : set CALL_PLAN;
    CustPlans  : CUST  →  set CALL_PLAN;
  Interface
    prov ip1:  AddNewCust(new_cust:CUST);
    prov ip2:  RemoveCust(cust:CUST_ID);
    prov ip3:  ModifyCust(cust:CUST_ID; new_rec:CUST);
    prov ip4:  AccessCust(cust:CUST_ID) : CUST;
    prov ip5:  EnterCallPlan(c:CUST_ID; p:CALL_PLAN);
  Invariant
    0 ≤ #Customers ≤ 1000000;
  Operations
    prov op1:
      Let   c:CUST;
      Pre   #Customers < 1000000 ∧ c ∉ Customers;
      Post  Customers' = Customers ∪ {c};
    prov op2:
      Let   c:CUST;
      Post  c ∈ Customers ⇒
            Customers' = Customers - {c};
    prov op3:
      Let   id:CUST_ID;
            c:CUST;
      Pre   id ∈ CustIds ∧ c ∉ Customers;
      Post  c ∈ Customers ∧ CustById(id) = c;
    prov op4:
      Let   id:CUST_ID;
      Pre   id ∈ CustIds;
      Post  result = CustById(id);
    prov op5:
      Let   id:CUST_ID;
            plan:CALL_PLAN;
      Pre   id ∈ CustIds ∧ plan ∈ CallPlans;
      Post  plan ∈ CustPlans(CustById(id));
  Map
    ip1 -> op1(new_cust -> c);
    ip2 -> op2(cust -> c);
    ip3 -> op3(cust -> id, new_rec -> c);
    ip4 -> op4(cust -> id);
    ip5 -> op5(c -> id, p -> plan);
end OnLineDB-2;
```

Fig. 12. *OnLineDB-2* is a candidate interface and behavior subtype of *OnLineDB*.

The *OnLineDB-2* component, shown in Fig. 12 is a candidate interface and behavior subtype of *OnLineDB* from Fig. 5. *OnLineDB-2* includes the already discussed features from *CallPlanDB* and *BoundedDB*. Additionally, it slightly changed operation specifications for *op1* and *op2* (corresponding, in this case, to interface elements *AddNewCust* and *RemoveCust*). *Op1* will not add a customer to the database if the database is already full. *Op2* does not require that the customer already be in the database; instead, it will check for the customer record and, if found, remove it. For *OnLineDB-2* to be an *int* **and** *beh* subtype of *OnLineDB*, the following must be true (from the schema *Oper_Conformance* in Fig. 10):

- *OnLineDB op1 pre* ⇒ *OnLineDB-2 op1 pre*
  `(c ∉ Customers) ⇒`
  `(#Customers < 1000000 ∧ c ∉ Customers)`
- *OnLineDB-2 op1 post* ⇒ *OnLineDB op1 post*
  `(Customers' = Customers ∪ {c}) ⇒`
  `(Customers' = Customers ∪ {c})`
- *OnLineDB op2 pre* ⇒ *OnLineDB-2 op2 pre*
  `(c ∈ Customers) ⇒ true`
- *OnLineDB-2 op2 post* ⇒ *OnLineDB op2 post*
  `(c ∈ Customers ⇒`
  `Customers' = Customers - {c}) ⇒`
  `(Customers' = Customers - {c})`

The first implication is not true. The left hand side (LHS) may be true even if the database is full, in which case the right hand side (RHS) is false. The second implication is true since LHS and RHS are the same. The third implication is true, since *OnLineDB-2*'s *op2* does not have any preconditions. Finally, the fourth implication is true. It has the form $(A \Rightarrow B) \Rightarrow B$, which is false if both A and B are false. However, B in our case will always be true because of the definition of set subtraction: if `c ∈ Customers`, `c` will be removed from the new value of the `Customers` set (`Customers'`); if this is not the case, `Customers` will simply remain the same.

Implementation_Conformance

Behavior_Conformance

Imp_Conf : *Component* ↔ *Component*

$$
\begin{aligned}
&\forall c1, c2 : Component \bullet \\
&\quad (c1, c2) \in Imp\_Conf \\
&\qquad \Leftrightarrow \\
&\quad (c1, c2) \in Beh\_Conf \wedge \\
&\quad (c1.invariant, c2.invariant) \in Logic\_Impl \wedge \\
&\quad (\forall o1 : Operation \mid o1 \in c1.operations \bullet \\
&\quad \exists o2 : Operation \mid o2 \in c2.operations \bullet \\
&\qquad (o2, o1) \in Oper\_Conf \wedge \\
&\qquad o1.implementation = o2.implementation)
\end{aligned}
$$

Fig. 13. Implementation conformance.

The first implication above is therefore the only one that violates the required relationship. It is because of this that *OnLineDB-2* is not an *int* **and** *beh* subtype of *OnLineDB*. Note that the architect may still decide to use *OnLineDB-2*, particularly since it is so closely related to *OnLineDB*, but must understand that *OnLineDB-2* cannot be substituted for *OnLineDB* in a correctness-preserving manner.

4.2.4. Implementation Conformance

Although useful in practice for evolving components, *implementation* conformance is not a particularly interesting relationship from a type-theoretic point of view. Implementation conformance may be established with a simple syntactic check if the operations of the subtype have identical implementations (both syntactically and semantically) as the corresponding operations of the supertype. Implementation conformance between two types thus also requires a behavioral equivalence between their shared operations, as shown in Fig. 13.

4.3. Type Checking a Software Architecture

In order to discuss type conformance of interoperating components, we must define an architecture that includes those components. There is no single, universally accepted set of guidelines for composing architectural elements. Instead, architectural topology depends on the ADL in which the architecture is modeled, characteristics of the application domain, and/or the rules of the chosen architectural style. We therefore had to make certain choices in specifying properties of an architecture:

- we model connectors explicitly, unlike, e.g., Darwin [12] and Rapide [10];
- we allow direct connector-to-connector links, unlike, e.g., Wright [1];
- finally, we assume certain topological constraints that are derived from the rules of the C2 style [25]: a component is attached to single connectors on its top and bottom sides, while a connector can be attached to multiple components and connectors on its top and bottom.

None of the above choices is required by our type theory. It is indeed possible to provide a definition of architecture that reflects any other compositional guidelines. However, these decisions were necessary in order to formally specify and check type conformance criteria.

Architecture

$components : \mathbb{P}\ Component$
$connectors : \mathbb{P}\ Connector$
$comp\_conn : Component \nrightarrow Connector$
$conn\_comp : Connector \leftrightarrow Component$
$conn\_conn : Connector \leftrightarrow Connector$
$Comm\_Link : Component \leftrightarrow Component$

$\mathrm{dom}\ comp\_conn = components$
$\mathrm{ran}\ comp\_conn = connectors$
$\mathrm{dom}\ conn\_comp = connectors$
$\mathrm{ran}\ conn\_comp = components$
$\mathrm{dom}\ conn\_conn = connectors$
$\mathrm{ran}\ conn\_conn = connectors$
$\mathrm{dom}\ Comm\_Link = components$
$\mathrm{ran}\ Comm\_Link = components$

$\forall\ c : Component;\ b : Connector \mid$
$\quad c \in components \land b \in connectors \bullet$
$\quad\quad (c, b) \in comp\_conn \Rightarrow (b, c) \notin conn\_comp \land$
$\quad\quad (b, c) \in conn\_comp \Rightarrow (c, b) \notin comp\_conn$

$\forall\ b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet$
$\quad (b1, b2) \in conn\_conn \Rightarrow (b1 \neq b2 \land (b2, b1) \notin conn\_conn)$

$\forall\ c1, c2 : Component \mid c1 \in components \land c2 \in components \bullet$
$\quad (c1, c2) \in Comm\_Link$
$\quad\quad \Leftrightarrow$
$\quad c1 \neq c2 \land$
$\quad (\exists\ b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet$
$\quad\quad ((c1, b1) \in comp\_conn \land$
$\quad\quad (b2, c2) \in conn\_comp \land$
$\quad\quad (b1, b2) \in conn\_conn^*)$
$\quad\quad\quad \lor$
$\quad\quad ((c2, b1) \in comp\_conn \land$
$\quad\quad (b2, c1) \in conn\_comp \land$
$\quad\quad (b1, b2) \in conn\_conn^*))$

Fig. 14. Formal definition of architecture.

The formal definition of architecture is given in Fig. 14. Connectors are treated simply as communication routing devices; therefore their definitions are omitted. Two components can interoperate if there is a communication link between them. This means that they are either on the opposite sides of the same connector or one can be reached from the other by following one or more connector-to-connector links (defined by the *Comm_Link* relation).

For example, in the architecture from Fig. 1, there is a communication link between *OnLineDB* and *NOSS_Mgr* components (via a single connector-to-connector link). There is also a link between *Transaction Consistency Checker* and *Customer Support Local Transaction State* components (different sides of the same connector). On the other hand, there is no communication link between *On-Line* and *NOSS* databases: they are attached below the same (top-most) connector; however, the *Comm_Link* relation mandates that they be on different sides of a connector, which reflects C2's communication rules.

Given this definition of architecture, it is possible to specify type checking predicates. As already discussed, components need not be able to fully interoperate in an architecture. The two extreme points on the spectrum of type conformance are:

- *minimal type conformance*, where at least one service (interface and corresponding operation) required by each component is provided by some other component along its communication links; and
- *full type conformance*, where every service required by every component is provided by some component along its communication links.

Minimal_Type_Conformance

Interface_Conformance
Behavior_Conformance
Architecture

$\forall\ c1 : Component \mid c1 \in components \bullet$
$\exists\ c2 : Component \mid c2 \in components \land (c1, c2) \in Comm\_Link \bullet$
$\quad (\exists\ ie1, ie2 : Int\_Element \mid$
$\quad\quad ie1 \in c1.interface \land ie2 \in c2.interface \bullet$
$\quad\quad\quad ie1.name = ie2.name \land$
$\quad\quad\quad ie1.dir = req \land ie2.dir = prov \land$
$\quad\quad\quad (ie1, ie2) \in Prm\_Conf \land$
$\quad\quad\quad (c1.int\_op\_map(ie1),$
$\quad\quad\quad\ c2.int\_op\_map(ie2)) \in Oper\_Conf)$

Full_Type_Conformance

Interface_Conformance
Behavior_Conformance
Architecture

$\forall\ c1 : Component;\ ie1 : Int\_Element \mid$
$\quad c1 \in components \land ie1 \in c1.interface \land ie1.dir = req \bullet$
$\exists\ c2 : Component;\ ie2 : Int\_Element \mid$
$\quad c2 \in components \land (c1, c2) \in Comm\_Link \land$
$\quad ie2 \in c2.interface \land ie2.dir = prov \bullet$
$\quad\quad ie1.name = ie2.name \land$
$\quad\quad (ie1, ie2) \in Prm\_Conf \land$
$\quad\quad (c1.int\_op\_map(ie1),\ c2.int\_op\_map(ie2)) \in Oper\_Conf$

Fig. 15. Type conformance predicates.

They are defined in Fig. 15. The predicates expressing the degree of utilization of a component's provided services in an architecture can be specified in a similar manner [17].

Depending on the requirements of a given project (reliability, safety, budget, deadlines, etc.), type conformance corresponding to different points along the spectrum may be adequate. What would be classified as a "type error" in one architecture may be acceptable in another. Therefore, architectural type correctness is expressible in terms of a percentage corresponding to the degree of conformance (per component or for the architecture as a whole).

4.3.1. Type Conformance and Off-the-Shelf Reuse

Before we can illustrate architectural type conformance with an example, we need to address another issue. Establishing type conformance brings up the question of how much a component may know about other components with which it will interoperate. Although magnified by our separation of provided from required component services, this issue is not unique to our type theory. Rather, it is pertinent to all approaches that model behavior of a type and enforce behavioral conformance.

To demonstrate behavioral conformance between two interoperating components, by definition one must show that a specific relationship holds between their respective behaviors. This relationship is one of several flavors of equivalence or implication, summarized in [29].

Establishing whether two components can interoperate includes matching the specification of what is expected by a required operation of one component against what another component's provided operation supplies. Behavior of an operation is modeled in terms of its interface parameters (in our approach, operation variables) and component state variables. A component may thus need to refer to state variables that

belong to another component in order to specify a *required* operation's expected behavior. However, doing so would be a violation of the "provider" component's abstraction. It would also violate some basic principles of component-based development:

- the designer may not know in advance which, if any, components will contain a matching specification for the required operation and, thus, what the appropriate (types of) state variables are. This is particularly the case when using behavior matching to aid component discovery and retrieval. For example, it is not reasonable to expect that a user of the *op1* operation in the *NOSS_Mgr* component from Fig. 5 would know that its behavior is expressed in terms of a function (*Numbers*) that, given an address, returns a set of phone numbers;
- large-scale, component-based development treats an off-the-shelf component as a black box, thereby intentionally hiding the details of its internal state. Having to explicitly refer to those details would require them to be exposed.

Existing approaches to behavior modeling and conformance checking have not addressed this problem. The problem does not apply to component subtyping: the designer must know all of existing component's details in order to effectively evolve it. Thus, those approaches that focus on behavioral subtyping (e.g., America [2], Liskov and Wing [11], and Leavens et al. [3]) do not encounter this problem. Zaremski and Wing [29] do address component retrieval and interoperability. However, their approach makes the very assumption that the designer will have access to a "provider" component's state (via a shared Larch trait [8]). Fischer and colleagues [4, 22] model components at the level of a single procedure. In order to be able to properly specify pre- and postconditions, they include all the necessary variables as procedure parameters. Thus, for example, the stack itself is passed as a parameter to the *push* procedure.

The solution to this problem we propose is based on two requirements arising from a more realistic assessment of component-based development:

- we do not have access to a "provider" component's internal state (unlike Zaremski and Wing's approach), and
- we cannot change the way many software components, especially in the OO world, are modeled (unlike Fischer et al.).

These two requirements result in an obvious third requirement:

- we must somehow refer to a "provider" component's state when modeling operations, even though we do not know what that state is.

This seeming paradox actually suggests our approach. The initial results of this approach are promising and we intend to further investigate its practicality.

We model a required operation as if we have access to a "provider" component's state. However, since we do not know the actual "provider" state variables or their types, we introduce a generic type, *STATE_VARIABLE*, which is a supertype of all basic types. Thus, variables of this type are essentially placeholders in logical predicates. When matching, e.g., a required and provided precondition, we attempt to unify (instantiate) each variable of the *STATE_VARIABLE* type in the required precondition with a corresponding state variable in the provided precondition. If the unification is possible and the implication (with all instances of *STATE_VARIABLE* placeholders replaced with actual variables) holds, then the two preconditions conform.

In the example from Fig. 5, *NOSS_Mgr* requires two services: *AddNewCust*, which is mapped to its operation *or1*, and *RemoveCust*, mapped to *or2*. OnLineDB provides operations with matching interfaces (as required by the type conformance predicates). Thus, to establish type conformance, we must now make sure that the operation pre- and postconditions are properly related. In the interest of space, we do so only for *or1*:

- *NOSS_Mgr or1 pre* $\Rightarrow$ *OnLineDB op1 pre*
  `(c ∉ cust_sv) ⇒ (c ∉ Customers)`
  In this case, `cust_sv` is instantiated with `Customers` and we have an implication of the form A $\Rightarrow$ A, which is obviously true.
- *OnLineDB op1 post* $\Rightarrow$ *NOSS_Mgr or1 post*
  `(Customers' = Customers ∪ {c}) ⇒`
  `(c ∈ cust_sv')`
  In this case, since `Customers` is the only state variable in the provided operation (*op1*), `cust_sv` is again instantiated with `Customers`, and the implication becomes
  `(Customers' = Customers ∪ {c}) ⇒`
  `(c ∈ Customers')`
  This implication is also true (if an item is added to a set, that item is an element of the set).

We have thus established that, at the least, minimal type conformance holds in the architectural interaction between *OnLineDB* and *NOSS_Mgr*.

4.4. Summary

This section defined and demonstrated with examples the major elements of our type theory: multiple subtyping relationships (Section 4.2) and type conformance (Section 4.3). Certain characteristics of our type theory are unique (e.g., separation of interface from behavior) and give rise to seemingly anomalous relationships when considered in isolation (e.g., supertype and subtype operation variable types depicted in Fig. 9). However, the type theory as a whole supplies mechanisms that prevent any such anomalies. For example, the *int_op_map* function constrains the actual use of operation variables with the types of interface parameters through which the variables are accessed. The desired relationship between a supertype's and subtype's operation variables is thus ensured.

5. CONCLUSIONS AND FUTURE WORK

Software architectures show great potential for reducing development costs while improving the quality of the resulting software. Architectures also provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools which also must be supported with specific techniques to achieve desired properties. This paper has outlined such a technique for supporting evolution of software components in a manner that preserves the desired architectural relationships and properties.

This technique is based on the recognition that, unlike PLs, software architectures need not always be rigid in establishing properties such as consistency and completeness. For example, it is not always the case that two components that share a communication link can actually communicate (e.g., due to

mismatched interfaces). At the architectural level, this can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into the implemented system, implementation-time decisions (e.g., communication via implicit invocation) may result in the loss of communication messages [15, 25], but still allow the rest of the system's architecture to perform at least in a degraded mode. Thus, informing the architect of the potential problem and leaving the decision up to the architect is often preferable to automatically rejecting the option.

We have already put many of these ideas into practice in the context of the C2 style and its accompanying ADL. We are currently developing a set of tools to support architectural subtyping, type checking, and mapping of architectural descriptions to the C2 implementation infrastructure [14]. We are also considering several existing theorem provers and model checkers to aid us specifically in establishing component invariant and operation pre- and postcondition conformance: NORA/HAMMR [22], Larch proof assistant (LP) [8], VCR [4], and PVS [20].

A number of issues remain items of future work. These include investigation of the applicability of our type theory for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, further research of issues in adapting and adopting legacy components into architectures using the subtyping approach, and automating the evolution of existing components to populate partial architectures.

6. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, Springer-Verlag, 1991.

[3] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.

[4] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-Based Software Component Retrieval Tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, New Orleans, LA, USA, December 1994.

[6] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

[7] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[8] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[9] D. Lea and J. Marlowe. Interface-Based Protocol Specifications of Open Systems Using PSL. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, Aarhus, Denmark, August 1995.

[10] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.

[11] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[12] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.

[13] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.

[14] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)* and *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.

[15] N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.

[16] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.

[17] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium*, Los Angeles, CA, April 1996.

[18] O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, Washington, D.C., USA, October 1993.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. To appear in *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 1998, Kyoto, Japan.

[20] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. *PVS:* Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, eds., *Computer-Aided Verification (CAV '96)*, volume 1102 of Lecture Notes in Computer Science, July/August 1996, Springer-Verlag.

[21] J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, vol. 3, num. 2, 1992.

[22] J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997.

[23] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.

[24] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science, Prentice Hall International, 1989.

[25] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.

[26] A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

[27] A. L. Wolf. Succeedings of the Second International Software Architecture Workshop (ISAW-2). *ACM SIGSOFT Software Engineering Notes*, January 1997.

[28] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.

[29] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.

On the Role of Connectors in Modeling and Implementing Software Architectures

Peyman Oreizy David S. Rosenblum Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
{peymano,dsr,taylor}@ics.uci.edu

Abstract

Software architectures are software system models that represent the design of a system at a high level of abstraction. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. A notable feature of many architectural models (and the languages used to express them) is their representation of communication concerns in explicit model elements, which are typically called connectors. However, there is little consensus yet in the software engineering community on the role of connectors in an architectural model, or even on the necessity of making them first-class model elements. In this paper we demonstrate the utility of explicit connectors in architectural models through a presentation and analysis of an architecture for a meeting scheduler system. We show how the functional abstraction provided by connectors contributes to the mobility, distribution and extensibility of the design, as well as its ability to sustain runtime structural change. Furthermore, we demonstrate how connectors encapsulate important aspects of inter-component communication, including the number and identity of communication recipients, the policy used to select these recipients, the choice of implementation technology for the communications, and architectural constraints on component composition.

1 Introduction

Software architectures are software system models that represent the design of a system at a high level of abstraction [8,10]. A software architecture typically focuses on the coarse-grained organization of functionality into components and on the explicit representation and specification of inter-component communication. Details at lower levels of abstraction, such as the selection of data structures and algorithms for individual modules, are the concern of later stages of design.

A notable feature of many architectural models (and the languages used to express them) is their representation of communication concerns in explicit model elements, which are typically called *connectors*. However, there is little consensus yet in the software engineering community on the role of connectors in an architectural model, or even on the necessity of making them first-class model elements.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973, by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-94-C-0218, and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research or the U.S. Government.

In this paper we demonstrate the utility of explicit connectors in architectural models through a presentation and analysis of an architecture for a meeting scheduler system. The architecture is constructed according to the C2 architectural style, whose connectors offer a number of important benefits over traditional approaches to system design [11].

2 A Traditional Design for a Meeting Scheduler

In the traditional approach to designing a meeting scheduler, the designer would first settle upon a well-known system organization (i.e., what a software architect would call an *architectural style*) and would then begin to allocate functional requirements to design elements. The design elements would be modules, subsystems, packages, subroutines—entities that the software architect would call *components*. As is common in the traditional approach, the designer would typically partition communication responsibilities among the different components, rather than defining them within an integrated design element that is separate from the component definitions.

For instance, one obvious design for the meeting scheduler is a client-server design. The initial design would typically involve a single server and several identical clients. The clients would provide a user interface through which meeting proposers and attendees can invite attendees, propose meetings, specify preference sets and exclusion sets, and specify resource requirements. The server would be responsible for *both* facilitating communication between the clients and for managing all of the information about meetings, meeting resources and availabilities.

Eventually, the basic client-server design might need to be scaled to support electronic meetings over wide-area networks. In this case, the single server would be evolved into a collection of federated servers that distribute the communications in some way and partition the information that they manage.

Another possible design for the meeting scheduler would use an event-condition-action style of interaction. This design may superficially resemble the client-server design, since transactions involving meetings, resources and availabilities would be directed to a server-like active database component. But communication would occur in a more asynchronous fashion than in the client-server design. As with the client-server design, an attempt to scale the design to a wide-area network would require redesign of the "server" component into a distributed active database.

While these designs represent reasonable solutions to the problem of distributed meeting scheduling from the viewpoint of traditional approaches to software design, they are nevertheless problematic from a number of viewpoints. First, while communication between the clients is a significant aspect of the requirements, the allocation of those requirements to design elements is constrained by the need to partition and encapsulate functionality across components. Thus, the design ends up containing no coherent, isolated representation of the communication that is to occur between components. This constraint leads then to the second problem, which is that the lack of explicit representation of communication leads to difficulties in the evolution of the system. One can easily imagine that the task of evolving a single, centralized server into a synchronized set of federated servers is a significant undertaking, and that the two server designs would have little in common and would provide limited opportunities for reuse.

3 Architectural Connectors

A *software architecture* represents software system structure at a high level of abstraction, and in a form that makes it amenable to analysis, refinement, simulation, and other engineering concerns [6]. The notion of a *connector* as an explicit architectural element is to be found in the earliest papers on software architecture [8]. The basic rationale for connectors is that they explicitly represent facilities for communication between components, which themselves represent the encapsulation of computation in a

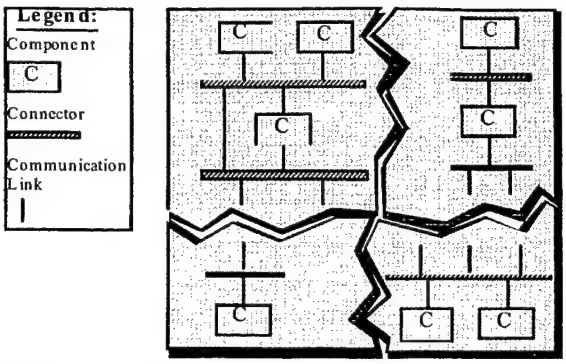


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

system. This rationale implies that connectors should be an explicit feature of any language for modeling software architectures. Such languages are called *architecture description languages*, or ADLs, and while the notion of a connector arguably has a strong intuitive appeal, not all designers of ADLs agree on their importance as a language feature. Indeed, in many well-known ADLs, inter-component connection is specified as part of the definitions of the affected components, rather than as an explicit, separate model element. Some of the better-known examples of such ADLs include Rapide [3,4] and Darwin [5]. Medvidovic calls such connection specifications *implicit connectors* [7].

Thus, architectural connectors provide a means for separating and making explicit the communication needs of a software system, and explicit connectors can be found in many ADLs, including Wright [1] and C2 [11]. Yet in attempting to formalize the notion of connectors as language elements, the designers of ADLs have struggled to maintain a sufficient distinction between components and connectors. In describing connectors and their behavior, it is particularly easy for the description to degenerate into something that makes connectors sound little different from components—they encapsulate functionality, they can encapsulate state, they interact with other components, and so on. Perhaps the clearest way of distinguishing the two is to view components as independent units of reuse, while connectors represent the *shared phenomena* of the components they connect.[1] In other words, components encapsulate specific functionality that can be used in many different applications, and they can execute autonomously. However, connectors exist only to serve the interaction needs of components. Furthermore, connectors do need to be just a modeling abstraction; they also provide benefit when they are explicit entities in the implementation.

In most ADLs that support explicit connectors, such as Wright, the connectors are defined with a finite and statically specified number of *ports*, or interaction points for components. This is perhaps one reason they resemble components so much, since the usual method of defining a component is to specify the contents of its interface, which typically contains a finite, predetermined number of interface elements (attributes, operations, exceptions, etc.). C2 supports a much different style of connector, which can best be understood in terms of the C2 architectural style.[2]

A C2 architecture is a hierarchical network of concurrent components linked together by connectors in accordance with a set of style rules. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a connector (see Fig. 1). The

[1] Jackson describes the notion of shared phenomena, which in this context technically refers to the shared phenomena of the *domains* of the components that a connector connects [2].

[2] For a more detailed discussion of the C2-style and its benefits see [11].

style does not place restrictions on the implementation language or granularity of the components. It does require that all communication between components occur by exchanging asynchronous messages through connectors. Since all message passing is done asynchronously, control integration issues are greatly simplified[3]. This remedies some of the problems associated with integrating components that assume that they are the application's main thread of control. Furthermore, components cannot assume that they will execute in the same address space as other components or share a common thread of control.

The layering of a C2 architecture is significant in that a component is only aware of the components above it, and *explicitly* utilizes their services by sending a request message. Communication with components below occurs *implicitly*. Whenever a component changes its internal state, it announces the change by emitting a notification message, which describes the state change, to the connector below it. The connector broadcasts these notification messages to all the components connected on its bottom side. Thus, notification messages provide an implicit invocation mechanism, allowing several components to react to the state change a component.

While connectors are modeled as explicit entities in several ADLs, they are not typically retained as explicit implementation entities. Instead, they are reified as procedure calls, data accesses, or linker instructions. C2 connectors are different in that they are explicit, runtime entities in the implementation responsible for routing messages between components. Additionally, the number of ports, or interaction points, on a C2 connector can change during runtime as components are added to and removed from it. The explicit and flexible nature of C2 connectors directly contributes to our ability to implement distribution, mobility, and runtime structural change. We describe these in the context of a C2-style architecture for the meeting scheduler.

4 A C2-Style Architecture for the Meeting Scheduler

In our C2-style architecture for the meeting scheduler, each user's display consists of three windows. A "meeting proposal" window allows scheduling of a new meeting. It lets the user specify the meeting agenda, select meeting attendees, and specify a meeting time and duration. A "schedule" window displays a list of the meetings the user has agreed to attend. An "invitation" window appears each time the user is invited to a meeting and allows the user to either accept or decline the invitation. If the user accepts the invitation, the meeting is added to the "schedule" window.

The C2-style architecture for the meeting scheduler is depicted in Fig. 2. White boxes represent components and gray boxes represent connectors. A line between a component and a connector represents a communication pathway between the two. The *Person ADT*, which manages an individual user's schedule, is created for each user in the system. Each user also has three artist components, each of which is responsible for displaying a graphical user interface for one of the three meeting scheduler windows described above. The meeting proposal artist (MP Artist) manages the "meeting proposal" window; the invitation manager artist (IM Artist) manages the "invitation" window; and the schedule manager artist (SM Artist) manages the "schedule" window. The *Local connector* routes the graphical rendering messages from the artists to the *graphics* component.

Consider a scenario in which four people are using the meeting scheduler and user 1 decides to schedule a new meeting with user 3 and user 4. User 1 uses the "meeting proposal" window managed by

[3] While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components.

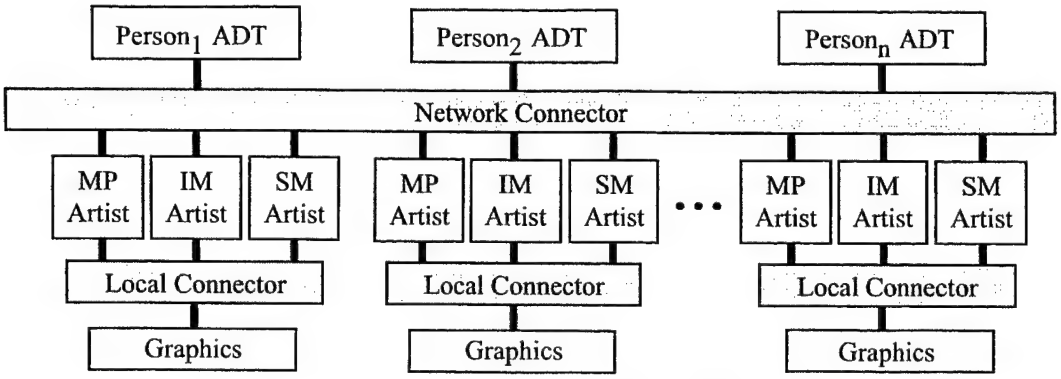


Fig. 2. The C2-style architecture for the meeting scheduler systems.

the *MP Artist* to specify the meeting time, location, and attendees. User 1's *MP Artist* then sends meeting invitation messages to *Person$_3$ ADT* and *Person$_4$ ADT*. Upon receiving the message, *Person$_3$ ADT* and *Person$_4$ ADT* each record the invitation in an internal data structure and emit a state change notification message. The notification is broadcast to all the components connected below *Network Connector*. User 3 and User 4's *IM Artists* react to the notification message from their *Person ADT* by displaying a new "invitation" window. If user 3 accepts the invitation, its *IM Artist* sends a message to *Person$_3$ ADT* confirming user 3's attendance at the meeting. Upon receiving the message, *Person$_3$ ADT* updates the user's schedule and emits a state change notification. User 3's *SM Artist* reacts to this notification by updating user 3's "schedule" window; the other meeting invitees' *SM Artists* react to this notification by noting user 3's acceptance in their "schedule" windows.

The functional abstraction provided by the connector facilitates the modeling and implementation of several unique properties of our design:

- *Mobility*—The *Network connector*, which is used to broadcast notifications about new meetings, and invitation acceptances and declinations, spans both multiple users and multiple machines. For instance, the *Network connector* routes meeting invitation messages from the *IM Artist* to the *Person ADTs* of the users that have been invited. The *Network connector* is the only entity in the architecture that knows the network location of individual components.

- *Distribution*—In contrast to other meeting scheduling applications such as Microsoft's Exchange Server and Sun Microsystems' Calendar Manager, there is no centralized meeting schedule server or ADT. Instead, each user has an individual meeting schedule ADT, namely the *Person ADT*, running on their local host. This ADT only stores information about the user's scheduled meetings. We can however emulate the centralized meeting scheduler design by placing all the *Person ADTs* on a single network host and by using a common database to store schedules.

- *Runtime structural change*—The runtime addition and removal of a user does not adversely affect other users. When a user comes online, their *SM Artist* broadcasts a message to all *Person ADTs* querying for meeting invitations that it missed while offline. The notifications resulting from the query are used to synchronize the user's display with that of the other users. It should be noted that supporting this type of runtime flexibility in a completely decentralized implementation has one drawback— a user will never learn of a meeting invitation unless at least one other meeting invitee is online at the same time.

- *Extensibility*—The loose coupling between the components afforded by the connectors enables different users to have differing implementations of the components, tailored to their particular

tasks and needs. This is possible as long as the different implementations adhere to the inter-component communication protocols governed by the connector.

5 Benefits Obtained from C2 Connectors

In other architectural styles, such as the client-server style, there is no coherent, isolated entity representing the communication between components[4]. As a result, decisions regarding inter-component communication are spread throughout individual application components. Connectors isolate a component's interfacing requirements from its functional requirements [9], thereby localizing decisions regarding communication policy and mechanism. C2 connectors go a step beyond other ADL connectors in that C2 connectors are explicit, runtime entities in the implementation. This enables C2 connectors to encapsulate:

1. the identity of the component receiving a particular message;
2. the number of components receiving a particular message;
3. the policy used to determine which components (from a set of eligible components) receive a message—If two or more components on a connector provide similar functionality, the connector may determine the most appropriate component to receive a given message. The decision may be based on communication latency, machine load, etc.;
4. the particular inter-process communication mechanism used for message passing—The connector can isolate the particular communication mechanism used to pass messages from one component to another (e.g., direct procedure calls, UNIX sockets, RPC, DCOM, CORBA);
5. the component's location in the network—Since components are not statically bound to one another, a component may migrate from one network node to another without having to notify other components;
6. the mapping from messages sent to message received—Since the connector acts as a conduit for communication, it can act as a domain translator between components;
7. the particular packaging and middleware technology used to implement components—Several different component packaging and middleware technologies exist for exposing the functionality of a component in a standard way. Connectors have the potential to act as a bridge between different technologies. Popular formats include COM, CORBA, Windows DLLs, and compiled Java byte-codes. The component packaging and middleware technology standardizes such things as how a component's methods are exposed for use; the invocation mechanism such as procedure calls, callbacks, and event loops; the proper order and types for passing method parameters; and the component's binary representation on disk. As long as a component adheres to the standard, the particular implementation language used by the component is inconsequential; and
8. the message to method mapping—If a component does not process C2 messages directly, the connector can provide a message to method mapping. This mapping, like the dynamic dispatch mechanism in Lisp, can potentially be altered during runtime. In fact, the binding does not have to be one-to-one. The connector may map a single message to several methods and combine the results in an appropriate manner.

[4] While one could attempt to model inter-component communication of such systems at the level of the computer network, the amount of detail and the potential discontinuity between the architectural model and the network topology obfuscates rather than elucidates the interactions.

As a result, architectural issues concerning these items may be considered separately from component functionality. This simplifies the behavioral model of components and enables us to more effectively partition the space of design issues.

We have successfully built C2 connectors that support items 1, 2, 4, and 5. Our C2 connectors also allow items 1 and 2 to be changed at runtime, enabling us to alter a system's structure during runtime. We are currently in the process of implementing a C2 connector that also supports item 7.

6 Discussion and Conclusion

In this paper, we have demonstrated the benefits of explicit connectors in architectural models. Connectors contribute to a separation of concerns in architectural modeling—they provide a convenient way to separate issues concerning component behavior from component interaction. This is especially important when constructing systems from reusable off-the-shelf components, since the designers of those components cannot anticipate all of the contexts in which the component is used. Our experience also demonstrates the utility of retaining the connectors in the implementation. The resulting functional abstraction contributes to the mobility, distribution and extensibility of the meeting scheduler, as well as facilitating runtime structural changes.

Finally, connectors provide a natural place for representing, analyzing and enforcing *architectural constraints*, which is a research topic of much current interest. In systems built from independently-constructed, off-the-shelf components, it is necessary to enforce constraints on how components are connected together and how they interact with each other. Furthermore, in many cases, it is most appropriate to evaluate and enforce such constraints at runtime. For instance, in the meeting scheduler architecture of Fig. 2, consider a Local Connector and the interactions it facilitates between the Artist components and the Graphics component. It is desirable to ensure that there is no overlap or duplication in the notifications that the Artist components send to the Graphics components, since otherwise the display presented by the Graphics component may become corrupted. The appropriate place to express and enforce this constraint is at the Local Connector, since the constraint applies to the set of components connected by the connector and not to any single component. A constraint is thus an invariant on the connector; it applies even when individual components are added to or removed from the connector. For example, the overlap constraint described above would prevent a newly added Agenda Artist from adversely affecting the user's display. If the constraints were associated instead with the Artists, then the constraints would have to be re-specified every time such a change is made to the configuration. Connection constraints such as these may be derivable from component interface specifications, but they still need to be evaluated and enforced in the context in which the components are used.

Acknowledgements

The C2-style architecture of the meeting scheduler described here is based on the one originally designed and implemented by Deborah L. Dubrow.

References

[1] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

[2] M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*: ACM Press/Addison-Wesley, 1995.

[3] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–355, 1995.

[4] D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, 1995.

[5] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

[6] N. Medvidovic and D.S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages", *Proc. USENIX Conference on Domain Specific Languages*, Santa Barbara, CA, pp. 199–212, 1997.

[7] N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, 1997.

[8] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[9] J.M. Purtilo, "The POLYLITH Software Bus", *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151–174, 1994.

[10] M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

[11] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

Critical Considerations and Designs for Internet-Scale, Event-Based Compositional Architectures

David S. Rosenblum[†] Alexander L. Wolf[‡] Antonio Carzaniga[‡]

[†]Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
dsr@ics.uci.edu

[‡]Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430
{alw,carzanig}@cs.colorado.edu

1 Introduction

A common architectural style for distributed, loosely-coupled, heterogeneous software systems is a structure based on event generation, observation and notification. A notable characteristic of an architecture based on events is that interaction among architectural components occurs asynchronously, thereby simplifying the composition of autonomous, independently-executing components that may be written in different programming languages and executing on varied hardware platforms.

There is increasing interest in deploying these kinds of distributed systems across wide-area networks such as the Internet. For instance, workflow systems for multi-national corporations, multi-site/multi-organization software development, and real-time investment analysis across world financial markets are just a few of the many applications that lend themselves to deployment on an Internet scale. However, deployment of such systems at the scale of the Internet imposes new challenges that are not met by existing technology.

In particular, the technology to support an event-based architectural style is well-developed for local-area networks (e.g., Field's Msg [7], SoftBench's BMS [1], ToolTalk [3] and Yeast [4]), but not for wide-area networks. One of these systems, Yeast, was built and studied by the first author while he was on the research staff at AT&T Bell Laboratories. Yeast is a general-purpose platform for building distributed applications in an event-based architectural style, and it supports event-based interaction quite naturally within local-area networks. However, its centralized-server architecture limits its scalability to wide-area networks, as does its limited support with respect to certain issues that are more important for wide-area networks than for local-area networks, such as naming and security. The experience with Yeast clearly demonstrates that these existing technologies are ill-suited to networks on the scale of the Internet, and that new technologies are needed to support the construction of large-scale, event-based software systems for the Internet.

We have been studying the problem of designing an Internet-scale event observation and notification facility that can serve as a platform for building wide-area distributed systems according to an event-based architectural style [8]. In this paper we briefly outline our achievements to date. In Section 2, we define more precisely what we mean by the notion of "Internet scale". In Section 3, we describe our design framework for an event observation and notification facility; this framework identifies a spectrum of design choices, which are organized according to seven models. In Section 4 we evaluate one existing technology, the CORBA Event Service, with respect to this design framework. We conclude in Section 5 with a discussion of our current work, which focuses on the design and analysis of architectures and algorithms for Internet-scale event notification.

2 Attributes of Internet Scale

The primary distinguishing characteristics of an Internet-scale computer network are the *vast numbers of computers* in the network and the *vast numbers of users* of these computers. An important related characteristic is the worldwide *geographical dispersion* of the computers and their users. As a consequence of geographical dispersion, it becomes necessary to address relativistic issues in multiple observations of the same event. For

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This work was also supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research or the U.S. Government.

instance, observers of two events occurring on opposite sides of the world may observe two different orders for those events. Additionally, an application requesting a notification about an event at roughly the same time as, but prior to, the occurrence of the event of interest may or may not be notified about the event.

At the scale of the Internet, the vast numbers of geographically-dispersed computers and users also have a much greater degree of *autonomy* than in local-area networks. Because of this autonomy, issues of resource usage are of greater concern, such as accounting for resource usage for observation and notification computations, placing limits on resource usage, and preventing misuse of resources or intrusiveness on others' usage of resources.

Related to the issue of autonomy is the *security* of the computers and users. Mechanisms and policies must be established that will allow Internet-scale event observation and notification to take place in a manner that is compatible with security mechanisms such as firewalls, and is consistent with the need to enforce access permissions and other protection mechanisms.

Finally, concerns related to *quality of service* obtain much greater visibility at the scale of the Internet. Because of network latencies, outages and other dynamically-varying network phenomena, an Internet-scale event observation and notification facility will have to cope with decreased reliability of observations and notifications, as well as decreased stability of the entities to be observed and notified.

As a consequence of Internet scale, it would be infeasible to employ many kinds of low-level mechanisms that are used to support event observation and notification in a local-area network, such as *broadcast mechanisms* and *vector clocks*. Broadcast mechanisms indiscriminately communicate event occurrences and notifications to all machines on a local network. Vector clocks involve piggybacking a vector timestamp onto each message exchanged between the communicating processes of an application, in order to aid the identification of causally-related events; the size of the timestamp is linear in the total number of processes in the application.

3 Design Framework

Implicit in event observation and notification is a timeline of basic activities, which occur in sequence:

1. *determination* of which events will be made observable;
2. *expression of interest* in an event or pattern of events;
3. *occurrence* of each event;
4. *observation* of each event that occurred;
5. *relation* of the observation to other observations to recognize the event pattern of interest;
6. *notification* of an application that its pattern of interest has occurred;
7. *receipt* of the notification by the application; and
8. *response* of the application to the notification.

To account for these activities, our design framework for Internet-scale observation and notification is organized around the following seven models, each of which focuses on a different domain of concern in the design:

1. an *object model*, which characterizes the components that generate events and the components that receive notifications about events;
2. an *event model*, which provides a precise characterization of the phenomenon of an event;
3. a *naming model*, which defines how components refer to other components and the events generated by other components, for the purpose of expressing interest in event notifications;
4. an *observation model*, which defines the mechanisms by which event occurrences are observed and related;
5. a *time model*, which concerns the temporal and causal relationships between events and notifications;
6. a *notification model*, which defines the mechanisms that components use to express interest in and receive notifications; and
7. a *resource model*, which defines where in the Internet the observation and notification computations are located, and how resources for the computations are allocated and accounted.

Each of these models has a number of possible realizations, which together define a seven-dimensional design space for Internet-scale event observation and notification facilities. Of course, these dimensions are not completely independent, because the models are interrelated in various ways. Because of these interrelationships, only a proper subset of the points in this space will correspond to adequate designs for Internet-scale facilities.

4 Evaluation of the CORBA Event Service

The Common Object Request Broker Architecture (CORBA) is a general-purpose, Internet-scale software architecture for component-based construction of distributed systems using the object-oriented paradigm [5,9]. The CORBA specification includes specifications for a number of Common Object Services, one of which is the CORBA Event Service [6]. The CORBA Event Service defines a set of interfaces that provide a way for objects to synchronously communicate event messages to each other. The interfaces support a *pull* style of communication (in which the *consumer* requests event messages from the *supplier* of the message) and a *push* style of communication (in which the supplier initiates the communication). Additional interfaces define *channels*, which act as buffers and multicast distribution points between suppliers and consumers. The TINA Notification Service is a similar service defined on top of the CORBA Event Service [10].

The CORBA Event Service lacks support for many aspects of event observation and notification defined in Section 3. The object model is the object model of CORBA, and an event is simply a message that one object communicates to another object as a parameter of some interface method. The specification of the CORBA Event Service does not define the content of an event message, so objects must be pre-programmed with "knowledge" about the particular event message structure that is to be shared between communicating suppliers and consumers. Given this view of events, a naming mechanism is unnecessary, as is an observation mechanism, and any attempt to identify patterns of events is the responsibility of the consumers of event messages. Timestamps can be associated with events, but the meaning of such timestamps is at the discretion of the objects exchanging the event messages. Being a message, an event is its own notification. Computational and other resource-related aspects of events are subsumed by those of CORBA as a whole.

In summary, an event as defined by the CORBA Event Service is nothing more than a parameter in a standard CORBA method invocation, with options available for multicasting and buffering message parameters. The application programmer wanting to use this service is still faced with the problem of how to locate events of interest, how to advertise new kinds of events, how to match patterns of events, and how to create and maintain networks of event channels to perform matching. Thus, the CORBA Event Service provides only a small subset of the capabilities needed in an Internet-scale event observation and notification facility.

5 Current Work

We have begun to design an improved event observation and notification facility. It is clear, based on our experience, that simply trying to scale a design intended for a local-area network is a flawed approach. The first step is therefore to formulate new architectures and related algorithms for event notification that are scalable to the Internet. Starting by adapting known Internet-scale architectures, such as that of Network News and the Domain Name Service, we simulate event observation and notification performance behavior in a variety of wide-area network scenarios. To date, four different architectures have been studied together with nine algorithms that implement both the recognition of event sequence patterns and the delivery of event notifications. The simulation is carried out by means of a network simulator that we have implemented. Our simulator allows us to configure a network model, and to define the event facility components, event generators, and event consumers. The simulator also accounts for computation and network resource usage on a per-host, per-process, and global-network basis.

Our initial target for the event observation and notification facility is the Software Dock [2], which is an agent-based system we are developing for Internet-scale distributed configuration management and deployment. The architecture of the Software Dock consists of *release docks*, representing producer sites, and *field docks*, representing consumer sites. The docks communicate through an event facility. The Software Dock is therefore a prototypical instance of a distributed compositional architecture. As one simple example of how the Software Dock would use an event facility, consider what happens when the producer of a system releases a bug fix. This would generate an event that results in notifications being sent to consumer sites interested in bug fixes for that system. The notification triggers an orchestration of agent activities that configure, retrieve, and install the fix.

As we gain experience in designing and constructing an Internet-scale event observation and notification facility, we will refine the models to incorporate lessons learned from our experience. A number of these refinements will likely be made to the observation and notification models, whose realizations will require careful engineering to ensure efficient and reliable operation on an Internet scale. Such refinements might involve the definition of a formal calculus of event operations that would support systematic optimization of the configuration of a network of observers, much in the same way that optimizations are applied to relational database queries in query languages such as SQL. Some operations that the calculus could support include generation, filtering, observation, notification, advertising, publication, subscription and reception.

References

[1] C. Gerety, "HP SoftBench: A New Generation of Software Development Tools", *Hewlett-Packard Journal*, vol. 41, no. 3, pp. 48–59, 1990.

[2] R.S. Hall, D. Heimbigner, A.v.d. Hoek, and A.L. Wolf, "An Architecture for Post-Development Configuration Management in a Wide-Area Network", *Proc. 1997 International Conference on Distributed Computing Systems*, Baltimore, MD, pp. 269–278, 1997.

[3] A.M. Julienne and B. Holtz, *ToolTalk and Open Protocols: Inter-Application Communication*: Prentice Hall, 1994.

[4] B. Krishnamurthy and D.S. Rosenblum, "Yeast: A General Purpose Event-Action System", *IEEE Transactions on Software Engineering*, vol. 21, no. 10, pp. 845–857, 1995.

[5] Object Management Group, "The Common Object Request Broker: Architecture and Specification", revision 2.0, July 1995.

[6] Object Management Group, "CORBAservices: Common Object Services Specification", formal/97-07-04, July 1997.

[7] S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, vol. 7, no. 4, pp. 57–66, 1990.

[8] D.S. Rosenblum and A.L. Wolf, "A Design Framework for Internet-Scale Event Observation and *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 344–360, 1997.

[9] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[10] Telecommunications Information Networking Architecture Consortium, "TINA Notification Service , July 1996.

Design of a Scalable Event Notification Service: Interface and Architecture

Antonio Carzaniga
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430, USA
+1 303 492 4463
carzanig@cs.colorado.edu

David S. Rosenblum
Dept. of Inf. and Computer Science
University of California
Irvine, CA 92697-3425, USA
+1 949 824 6534
dsr@ics.uci.edu

Alexander L. Wolf
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430, USA
+1 303 492 5263
alw@cs.colorado.edu

ABSTRACT
Event-based distributed systems are programmed to operate in response to events. An event notification service is an application-independent infrastructure that supports the construction of event-based systems. While numerous technologies have been developed for supporting event-based interactions over local-area networks, these technologies do not scale well to wide-area networks such as the Internet. Wide-area networks pose new challenges that have to be attacked with solutions that specifically address issues of scalability. This paper presents Siena, a scalable event notification service that is based on a distributed architecture of event servers. We first present a formally defined interface that is based on an extension to the publish/subscribe protocol. We then describe and compare several different server topologies and routing algorithms. We conclude by briefly discussing related work, our experience with an initial implementation of Siena, and a framework for evaluating the scalability of event notification services such as Siena.

1 INTRODUCTION

The *event-based* architectural style is well established and widely used. Several classes of applications adopt an event-based architecture, including integrated development environments, workflow and process support systems, software deployment systems, graphical user interfaces, network management tools, and security monitors. In this style, components are programmed as reactive objects that perform actions in response to certain events. Such style is particularly suitable for applications that are reactive by nature, such as network and security monitors, and also for systems that integrate heterogeneous components and thus require loosely coupled interaction.

The connectivity provided by wide-area networks such as the Internet offers even stronger motivation for using an event-based architecture. New applications can be designed that take advantage of the vast number of information sources available on-line. Examples are stock market analysis tools and data mining and indexing tools. Also, in the context of a wide-area network, existing applications can be integrated at a much higher scale; for example, workflow systems can be federated for companies that have multiple distributed development sites or even across corporate boundaries.

The common infrastructure underlying event-based systems is the *event service*. An event service is a general-purpose facility that provides for *observation* and *notification* of events among distributed objects. Numerous technologies that realize an event service have been developed and effectively used for quite a long time. However, most of them target local-area networks. Extending the support of an event service to a wide-area network creates new challenges and trade-offs. Not only does the number of objects and events grow tremendously, but also many of the assumptions made for local-area networks, such as, low latency, abundant bandwidth, homogeneous platforms, continuous reliable connectivity, and centralized control, are no longer valid.

Some technologies address issues related to wide-area services. Among them, are new technologies such as Tibco [8] that specifically provide an event service, but also, more mature technologies such as the USENET news infrastructure, IP multicasting, the Domain Name Service (DNS), that, although not explicitly targeted at this problem domain, represent potential or partial solutions. The main problem with all of these technologies is that they are either specific to some application domain or not flexible enough to be usable as a generic infrastructure for event-based applications.

This paper presents Siena, a project directed towards the design and implementation of a scalable general-purpose event service. The contributions of this work are a formal definition of an event service that combines expressiveness with scalability together with the design and implementation of the architectures and algorithms that realize this event service as a distributed infrastructure. One obvious issue that we must face in this research is the evaluation of the solu-

tions that we propose. To this end, we have performed systematic simulations of our architectures and algorithms in several network scenarios.

The following section gives the basics of the Siena event service. The paper then continues in Section 3 with a formal definition of the interface and the semantics of the event service. The architectures and algorithms that realize the service are presented in Section 4. Section 5 provides an overview of some related systems and research topics. Our evaluation effort and our experience with a prototype is presented in Section 6. We then conclude in Section 7 with some directions for future work and additional analysis and evaluation.

2 EVENT SERVICE

An *event service* is a dispatcher of event notifications. Applications that use the event service can be *interested parties*, i.e., event consumers, or *objects of interest*, i.e., event generators, or both. The dispatching is regulated by advertisements, subscriptions, and publications.

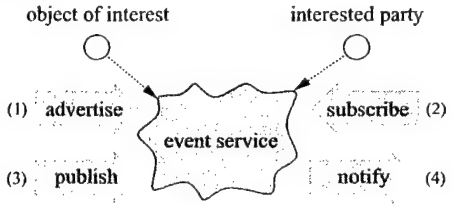


Figure 1: Event service

Figure 1 shows the high-level architecture of an event service. Informally, objects of interest specify the events they intend to publish by means of *advertisements* (1), while interested parties specify the events they are interested in by means of *subscriptions* (2). Objects of interest can then publish *notifications* (3), and the event service will take care of delivering the notifications to the interested parties that subscribed for them (4). The terms used in this paper, in particular the terms *notification*, *object of interest*, and *interested party*, follow the framework proposed in [13].

Without loss of generality, we will always assume that objects of interest are "active", i.e., they autonomously publish event notifications. Passive objects, such as files, can participate in an event-based interaction by means of other active objects that act as *proxies* and that notify events on behalf of the passive objects. This distinction is similar to the one made in JEDI [2]. In any case, the passive object will not be considered in the models.

Event Servers

The event service can be realized by connecting many *events servers*. An application contacts the event service via one event server also referred as its *access point*. (see Figure 2).

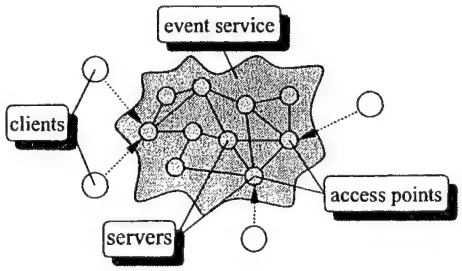


Figure 2: Internal architecture of the event service

Identifiers and Handlers

In order for interested parties, objects of interest, and event servers to communicate, a *naming* scheme must be adopted whereby objects can be uniquely identified. A *handling* scheme must also be adopted so that objects can be contacted using appropriate communication protocols.

The Siena event service adopts the generic URI [1] form for both its naming and handling scheme. This means that every object has a URI associated with it that defines both the *identity* of that object and the *handler* used by the event service to deliver a notification to that object. For example, if the URI *mailto:carzanig@cs.colorado.edu* identifies an object, then *mailto:carzanig@cs.colorado.edu* is both the unique name of that object and the method that the event service uses to communicate with that object. In this case, in order to send a notification to that object, the event service will send an e-mail message to *carzanig@cs.colorado.edu*.

The event service recognizes the most common URI schemas, including *mailto* and *http*, and thus implements the communication protocols implied by each schema. The implementation of the event service defines and maintains the URIs corresponding to event servers, however it does not directly assign or maintain URIs for interested parties or objects of interest. Such URIs are provided and operated by clients themselves. This means that if a client identifies itself as *mailto:carzanig@cs.colorado.edu*, then the event service will simply assume that the mailbox *carzanig@cs.colorado.edu* exists and is directly accessible.

3 INTERFACE AND SEMANTICS OF THE EVENT SERVICE

The Siena event service exports the following main functions:

publish(notification n)
subscribe(URI subscriber, pattern p) unsubscribe(URI subscriber, pattern p)
advertise(URI publisher, filter f) unadvertise(URI publisher, filter f)

In the following subsections we present the syntax and the semantics of these functions by formally defining *notifications*, *filters*, and *patterns* and their role in every function. We then present a formal definition of the semantics of the event service showing how it can affect scalability.

Notifications, Filters, and Patterns
An event notification is a set of attributes in which each attribute is a triple: *attribute* = (*name*, *type*, *value*). For example, the notification displayed in Figure 3 represents a stock price variation event.

string	*event*	= *finance/exchanges/stock*
time	*date*	= *Mar 4 11:43:37 MST 1998*
string	*exchange*	= *NYSE*
string	*symbol*	= *DIS*
float	*prior*	= *105.25*
float	*change*	= *-4*
float	*earn*	= *2.04*

Figure 3: Example of a notification

In an event notification, attributes are uniquely identified by their name. Attribute types belong to a predefined set of types. A fixed set of operators is also defined. Types and operators are an integral part of the event service definition. We do not a give a precise definition for the types and operators here, but instead simply assume those defined in modern programming languages. If α is an attribute of a notification, $\alpha.name$, $\alpha.type$, and $\alpha.value$ denotes its name, type, and value respectively.

Filters
An *event filter*, or simply a *filter*, defines a class of event notifications by specifying a set of attribute names and types and some constraints on their values.

string	*event*	*=	*finance/exchanges/**
string	*exchange*	==	*NYSE*
string	*symbol*	==	*DIS*
float	*change*	<	*0*

Figure 4: Example of an event filter

Figure 4 shows a filter that selects negative stock price variations for a specific stock on a specific exchange. More formally, a filter is made of a set of *attribute filters*. Each attribute filter specifies a name, a type, a boolean binary operator, and a value for an attribute: *attr-filter* = (*name*, *type*, *operator*, *value*). In an event filter, there can be more than one attribute filter with the same name. For an attribute filter α, $\alpha.match\_op(operand_1, operand_2)$ denotes the application of the operator defined by α to $operand_1$ and $operand_2$.

Patterns
A *pattern* of events is defined by combining a set of event filters using filter *combinators*.

string	*event*	*=	*finance/exchanges/**
string	*symbol*	==	*MSFT*
float	*change*	<	*0*
	and then		
string	*event*	*=	*finance/exchanges/**
string	*symbol*	==	*NSCP*
float	*change*	>	*0*

Figure 5: Example of a pattern of events

An example of a pattern that combines two filters into a sequence is shown in Figure 5. More formally, an event filter is itself a pattern, and any two patterns can be combined to form another pattern by means of a combinator. Intuitively, while a filter selects one event notification at a time, a pattern can select several notifications that *together* match an *algebraic combination* of filters.

We say that a pattern is *simple* when it contains only one event filter. Also, since subscriptions submit patterns to the event service, we say that a subscription is *simple* when it requests a simple pattern or *compound* when it requests a pattern with two or more filters.

For the purpose of this paper, we will only discuss the *and then* or *sequence* combinator that construct patterns matching a temporal sequence of events.

Compatibility Relations
In order to give the precise semantics of the event service, we must introduce and define the concept of *compatibility* between notifications and subscriptions, and between subscriptions and advertisements. The compatibility between notifications and subscriptions defines the semantics of subscriptions and comes into play because the main job of the event service is to decide whether or not notifications that are published match any subscription submitted by an interested party. In case a notification matches some subscriptions, the event service routes the notification towards all the interested parties that posted such subscriptions. The compatibility between subscriptions and advertisements is also important because, in setting up the routing information, the event service takes advertisements into account to see if they

are relevant to any subscription. The compatibility between subscriptions and advertisements subsumes a relation between notifications and advertisements that defines the semantics of advertisements.

The following sections define what it means for a notification to be compatible with a subscription and for a subscription to be compatible with an advertisement. Initially we consider only simple subscriptions (i.e., event filters) and then extend the compatibility relations to compound subscriptions.

Notifications vs. Subscriptions
Let $\mathcal{N}$ be the domain of notifications and $\mathcal{S}_0$ the set of all the simple subscriptions. We define the following binary relation:

$$IsCompatible_N^S \subseteq \mathcal{N} \times \mathcal{S}_0$$

For brevity, we represent the relation $IsCompatible_N^S$ with the symbol '$\sqsubset_N^S$'. When a notification n is compatible with a subscription s, we also say that s *covers* n, and we denote with $N(s) \subseteq \mathcal{N}$ the set of notifications n covered by s.

We define the semantics of $\sqsubset_N^S$ by defining $N(s)$ as follows:

$$\begin{aligned} N(s) = \{n \in \mathcal{N} : \forall \alpha_s \in s : \exists \alpha_n \in n : \\ \alpha_s.name = \alpha_n.name \wedge \alpha_s.type = \alpha_n.type \\ \wedge \alpha_s.match\_op(\alpha_n.value, \alpha_s.value)\} \end{aligned} \quad (1)$$

This mandates that all attributes in the subscription appear by name in the notification and that they match by type and value. The notification can also contain other attributes that are not specified in the subscription.

Subscriptions vs. Advertisements
We first define the semantics of advertisements similarly to what we have done in the previous section for subscriptions. Let $\mathcal{A}$ be the domain of advertisements and $a \in \mathcal{A}$ an advertisement. We define the set of notification defined (or *covered*) by a:

$$\begin{aligned} N(a) = \{n \in \mathcal{N} : \forall \alpha_n \in n : \exists \alpha_a \in a : \\ \alpha_n.name = \alpha_a.name \wedge \alpha_n.type = \alpha_a.type \\ \wedge \alpha_a.match\_op(\alpha_n.value, \alpha_a.value)\} \end{aligned} \quad (2)$$

This says that an advertisement covers all the notifications that have a set of attributes *included* (present by name and matching by value) in the set of attributes of the advertisement.

Given the definition of $N(a)$ we can easily define $IsCompatible_S^A$ ($\sqsubset_S^A$ for short), the compatibility relation between subscription and advertisements:

$$\sqsubset_S^A \subseteq \mathcal{S}_0 \times \mathcal{A}$$

Intuitively, the compatibility between a subscription s and an advertisement a corresponds to the relation between the two sets of notifications defined by s and a respectively, thus:

$$s \sqsubset_S^A a \Leftrightarrow N(a) \cap N(s) \neq \emptyset \quad (3)$$

This says that a subscription s is compatible with an advertisement a whenever the set of notifications defined by a, $N(a)$, includes one or more notifications that are also covered by s. When a subscription s is compatible with an advertisement a, we also say that a *covers* s.

Semantics of the Service
In this section we discuss the behavior of the event service in response to advertisements, subscriptions, and notifications. We have studied and implemented two alternative semantics:

- *subscription-based*, and
- *advertisement-based.*

These two behaviors define two *different* event services. The reason to present both and not to make a definite choice here is that these two semantics impose different requirements upon the implementation of the event service, resulting in different architectures with different degrees of scalability. At this point, we do not have enough experience in using the event service to know which one is more suitable, flexible, and scalable. It might also make sense to provide both of them and let the user choose which one works best for each particular situation.

Subscription-based Event Service
In the *subscription-based* event service, only subscriptions determine the semantics of the service. Advertisements *may* be used by the event service (e.g., to optimize the routing of subscriptions), but they are *not required.* The event service will guarantee the delivery of a notification to all interested parties that have subscribed for it. Referring to the compatibility relation between notifications and subscriptions, the event service will deliver a notification n to an interested party X *if and only if*:

1. X subscribes for s; *and*
2. $n \sqsubset_N^S s$.

Advertisement-based Event Service
In the *advertisement-based* event service, *both* advertisements and subscriptions are used. In particular, advertisements are used to make notifications visible to all the participants of the event service. More

specifically, the event service will guarantee the delivery of a notification n posted by object Y to interested party X *if and only if*

1. Y advertises a;
2. X subscribes for s;
3. $s \sqsubset_S^A a$; *and*
4. $n \sqsubset_N^S s$.

Note that if an interested party X sends a subscription s' that covers n, but Y has never posted any advertisement a that covers s', then the event service will not guarantee the delivery of n to X.

Patterns

So far we have discussed the semantics of the event service for *simple* subscriptions, i.e., for subscriptions that are composed of one event filter. However, both the subscription-based and the advertisement-based semantics can be easily extended to incorporate patterns.

As described above, patterns are defined by *pattern filters*, which are expressions whose elementary terms are simple filters. Thus, a subscription to a pattern filter can be logically viewed as a set of separate subscriptions to all the elementary components of that pattern filter plus a monitor that assembles sequences of notifications, each one matching one of the elementary components according to the semantics of the combinators. Thus, the event service will guarantee the delivery of a pattern of notifications matching an event filter only if it can guarantee the delivery of all the elementary components of the filter. Note that, from this definition of the semantics of patterns, the delivered pattern of notifications contains the *first* notification matching each elementary component.

Comments on the Semantics of the Event Service

The rationale behind the two semantics and their extensions to patterns is to define an event notification service that (1) behaves in an intuitive and useful way, and (2) allows for an efficient and scalable realization. In this paper, we do not explore the domain of applications that would make use of an event service, so we rely on our previous research and experience to justify the first item. Instead, we will elaborate more on the second item by showing how the information provided by advertisements and subscriptions with the given semantics can be effectively used to direct the communication between event servers in an efficient way.

Timing and *quality of service* are important, but they're not covered in details in this paper. Timing issues might arise when considering unsubscriptions and unadvertisements. For example, an interested party may send an unsubscription when some notifications have already been sent to it. In that case, the interested party will probably receive undesired notifications. Other timing issues regarding the ordering of notifications and thus pattern recognition can arise depending on the topology and latency of the network. For the time being we will assume that the event service is able within a finite time to shuffle notifications so that they are sent (and received) in the correct temporal sequence.[1]

By quality of service we refer to a number of non-functional properties that do not directly affect the semantics, but that are nonetheless of fundamental importance for the practical realization and usage of the event service. A number of other interface functions will be added to deal with quality of service settings such as authentication and security, and transactional communications.

Rationale: Expressiveness vs. Scalability

The rationale for our formal definition of notifications, filters, patterns, and compatibility relations goes beyond a clear specification of the semantics of the event service. The realization of the event service by means of distributed event servers, requires to disseminate some information concerning subscriptions and advertisements among event servers in order to control the flow of notifications towards interested parties. In the distribution of this information, the compatibility relations together with other similar relations between filters ($\sqsubset_S^S$ that defines the compatibility of two simple subscriptions and $\sqsubset_A^A$ that works for two advertisements), play a fundamental role. In fact, similarly to the optimization of queries in a database, using the compatibility relations, the event service can optimize the deployment of filter- and pattern-matchers to minimize the usage of communication and computation resources.

Thus, for the practical realization of the event service and for its scalability, it is essential that these relations can be efficiently implemented. The relations that pose significant problems are clearly the ones that involve two filters (e.g., $\sqsubset_S^A$); in fact, computing $n \sqsubset_N^S s$ is just a matter of applying the filter defined by s to n, which involves computing a conjunction of simple predicates evaluated for a particular instance of their independent variables. On the other hand, compar-

[1] This assumption would require the existence of a global clock, an upper bound for the network latency and the network diameter, and sufficiently big communication buffers. Note that while these latter requirements can pose serious engineering trade-offs, the availability of high-resolution GPS services makes the first assumption very reasonable for most practical applications.

ing two filters, to verify $s \sqsubset_S^A a$ is equivalent to verifying the implication between two expressions of predicates for every possible notification.

Even in our particular case in which filter expressions are conjunctions of simple predicates, this problem can be very hard to solve depending on the nature of types and operators that constitute the simple predicates. Given an attribute filter $f_1 = (N, T, Op, V)$ of name N, type T, operator Op and value V, and another attribute filter $f_2 = (N, T, Op', V')$ having the same name and type plus operator Op' and value V', we want to be able to decide whether or not the first filter implies the second:

$$(f_1 \Rightarrow f_2) \Leftrightarrow \forall x \in T : Op(x, V) \Rightarrow Op'(x, V')$$

Good operators are those that define equivalence relations and order relations on totally ordered sets. The usual set of basic types found in a modern programming language (numbers, strings, chars, booleans, etc.) and the usual operators (equality, inequality, regular expression match for strings), satisfy this constraint and also constitute a quite expressive vocabulary for filters.

Other systems adopt different notification models and different filtering capabilities. As a consequence, they realize different degrees of expressiveness and scalability. Section 5 comments on some of these choices with respect to the expressiveness/scalability spectrum.

4 TOPOLOGIES AND ALGORITHMS

The Siena event service is architected as a distributed system. This section presents some alternative realizations in which many *event servers* cooperate to provide a network-wide event service.

Server Topologies

Hierarchical Server Topology

A natural way of connecting event servers is according to a *hierarchical* topology; for instance, this is the topology of the distributed implementation of the JEDI event dispatcher [2]. As shown in Figure 6, each server in a hierarchical topology has a number clients that can be either normal objects of interest or interested parties or other event servers. In addition to these connections, a server could also have a special connection to a parent server (the only outgoing arrow).

It is important to note that in this topology, a server does not distinguish between other servers and its clients, and thus it treats those servers as clients. Practically, this means that a 'parent' server will be able to receive notifications, subscriptions, and advertisements from all its clients, but it will send only notifications to its clients.

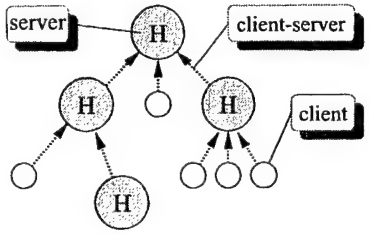


Figure 6: Hierarchical server topology

Acyclic Peer-to-Peer Server Topology

In the acyclic peer-to-peer topology, servers communicate with each other as peers, thus allowing a bidirectional flow of subscriptions and advertisements as well as notifications. Figure 7 shows an acyclic peer-to-peer topology of servers. Once again, notice the different kinds of communication occurring between clients and servers and among servers.

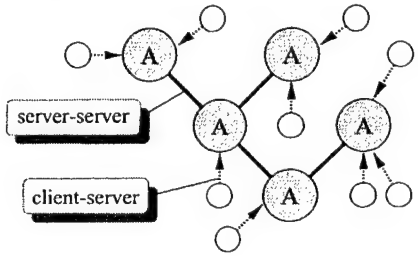


Figure 7: Acyclic peer-to-peer server topology

Generic Peer-to-Peer Server Topology

The generic peer-to-peer topology allows the same type of server-to-server communication introduced by the acyclic peer-to-peer, but in addition to that, it allows any pattern of connections between servers (see Figure 8).

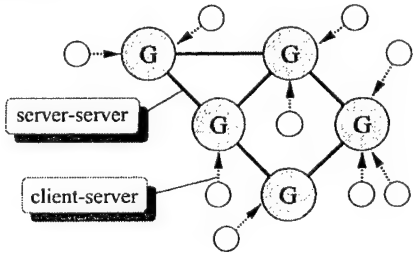


Figure 8: Generic peer-to-peer server topology

Routing Techniques

Once connected, server must exchange notifications, subscriptions, and advertisements to realize the service. This section presents the algorithms that disseminate the proper information throughout the network of servers making sure that notifications are correctly delivered. Note that a network of servers imple-

menting the event service is logically equivalent to a network of routers connecting sub-nets and realizing multicast routing. In fact, the algorithms presented here are very similar in principle to a combination of the Internet Group Management Protocol (IGMP [6]) and a reverse-path multicast routing algorithm [5, 4]. More details on the similarities and differences with network-level multicasting can be found in Section 5.

We have defined two classes of algorithms:

subscription forwarding: This technique uses subscriptions to set the paths for notifications. Every subscription is stored and forwarded from the originating server to all the servers in the network so to form a tree that connects the subscriber to all the servers in the network. When an object publishes a notification that matches that subscription, the notification is routed towards the subscriber following in reverse the path put in place by the subscription;

advertisement forwarding: This technique uses advertisements to set the paths for subscriptions, which in turn set the paths for notifications. Every advertisement is forwarded throughout the network, thereby forming a tree that reaches every server. When a server receives a subscription, it propagates the subscription in reverse along the path to the advertiser, thereby *activating* the path. Notifications are then forwarded only through the *activated* paths.

There exists also the degenerate case of broadcasting notifications, which we do not take into consideration. Unsubscriptions and unadvertisements are handled in a similar way to undo the effect of the corresponding subscription or advertisement. As suggested by their names, subscription forwarding and advertisement forwarding implement the subscription-based and advertisement-based semantics respectively. There are two main optimization strategies for saving communication and computation resources that can be pursued by applying these two algorithms, they are:

1. *applying filters and matching patterns upstream*: this means filtering notifications and assembling patterns of notifications as close as possible to publishers (see Figure 9);
2. *replicating notifications downstream*: this means multicasting notifications as close as possible to subscribers (see Figure 10).

The broadcasting or *flooding* process that characterizes both subscription forwarding and advertisement

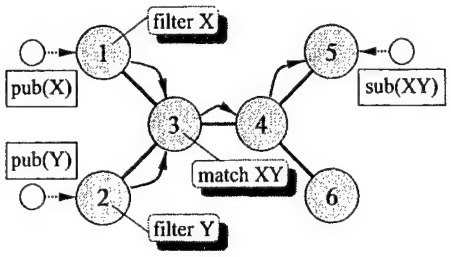


Figure 9: Applying filters and patterns upstream

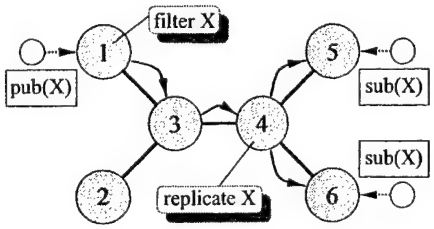


Figure 10: Multicasting of notifications downstream

forwarding creates minimal spanning trees for each source. The realization of this process depends upon the underlying topology of servers. The solution is trivial in the case of acyclic topologies (i.e., hierarchical and acyclic peer-to-peer), but it requires additional data structures and protocols for the generic graph topology [3].

In propagating requests, servers maintain tables of subscriptions and/or advertisements. When an event server receives a new request, say a subscription, that is already *covered* by a previously served one, the server simply adds the subscriber to the local list and no other action is taken. If no such subscripition is present in the tables, the new request is added to the table and propagated. This allows to prune of entire subtrees in the propagation. For example, in the scenario of Figure 10, when server 4 receives the forward of a subscription for X from server 5 for the first time, it propagates it to server 3, and then all the way towards server 1. However, when server 6 sends the same subscription to server 4, server 4 stops the flooding.

Servers perform other types of optimizations too. For example, in the advertisement forwarding algorithm, when a server receives an advertisement from one of its neighbor servers for which there exist matching subscriptions, the server forwards all these subscriptions to the advertiser. In doing that, the server tries to merge the batch of subscriptions into a smaller number of more generic subscriptions. Again, this can be done thanks to the simple structure of filters and the predictable nature of predicates and types that can be

used in filters.

For the recognition of patterns, event servers try to assemble patterns from smaller sub-patterns or single notifications that are already "available". To do this, servers rely on their tables of advertised patterns. In short, when a server receives a subscription that requests a pattern, it looks up the table of advertised filters trying to break up that sequence into smaller available filters or patterns. If sub-patterns that together form the target sequence have been advertised by local clients or neighbor servers, the server dispatches subscriptions for every one of these parts and starts up a monitor that recognizes the requested sequence from the sub-sequences.

Whenever possible, a server will push the recognition of entire sub-sequences towards the sources of their components. For example, in the scenario of Figure 9, server 5 notices that the sequence XY can be broken into X and Y and that both these parts are available from the same server (4), thus server 5 simply forwards the subscription for the entire sequence. Server 4 in turns does exactly the same thing forwarding the subscription to server 3. Server 3 notices that X and Y are advertised by two different sources, thus it sends the two separate simple subscriptions and start up the monitor.

Once again notice how the compatibility relations are crucial in every step of the forwarding techniques. Also notice that the way that the compatibility relations define the semantics of the event service is motivated by the forwarding techniques. In particular, the constraints posed by the advertisement-based semantics make it possible for event servers to maintain advertisements tables that are necessary for decomposing and optimizing pattern recognition.

5 RELATED WORK

In this section we provide a brief survey of technologies that we believe are tightly related to the problem of wide-area event notification, either because they attack the same problem or because they provide important pieces of solutions.

Internet Basic Technology

A number of Internet technologies are worth mentioning because, if nothing else, they indeed realize services on a wide-area network. Thus, even if none of them is geared towards an event notification service, it might be worthwhile to borrow their ideas vis-a-vis scalability.

Domain Name Service

DNS is a scalable network service that is realized in a distributed manner. The valuable idea that we can borrow from DNS is its hierarchical architecture. The reason why the hierarchical architecture works so well for DNS is that it maps very well onto the data that it manages. In fact, the space of domain names and the space of IP addresses are hierarchical themselves and the mapping between them preserves a lot of the hierarchical properties. Unfortunately, the space of notifications doesn't exhibit any hierarchical structure and, even if we decided to force this type of structure (e.g., by defining a mandatory well known attribute and a hierarchical set of values for it), this would not naturally map onto a hierarchical location of objects. Another differentiator is the essential read-only nature of the DNS service, which does not apply much to event notification services.

USENET News

The USENET News system with its main protocol NNTP [9] is perhaps the best example of a scalable user-level one-to-many communication facility. USENET News messages are modeled after e-mail messages, yet they provide additional information (headers) that can be used by NNTP commands to direct their distribution. NNTP provides both client-server and server-server commands. The topology of news servers is very similar to (and in fact inspired) the acyclic peer-to-peer topology.

The main problem with USENET News and NNTP that limits usability as an event service is that the selection mechanisms are not very sophisticated. At the protocol level, messages are filtered based only on their newsgroups and on their date. The newsgroup name space is organized in a hierarchy, and the protocol allows wild-card expressions over group names. Although group names and sub-names reflect the general subject and content of messages, the filter that they realize is too coarse-grained for most users and definitely inadequate for a general-purpose event service. This results in unnecessary transfers of entire groups of messages. The service is thus scalable but still quite heavyweight, and the time frame of news propagation ranges from hours to days.

IP Multicast

MBone is a network-level infrastructure that realizes an efficient one-to-many communication service. An MBone or multicast IP address is a virtual address that corresponds to a *group* of hosts. Hosts can join or leave a group at any time. An IP packet addressed to a host group will be delivered to all the hosts in that group. IP multicast per se is a connectionless best-effort (unreliable) service. A reliable transport layer can be implemented on top of IP multicast.

We consider the IP multicast infrastructure and its routing algorithms to be the most important technology related to the Siena event service. IP multicast

may be used as an underlying transport mechanism for notifications and the ideas developed for routing multicast packets can be adapted to solve the problem of forwarding notifications in an event service. But the IP multicast infrastructure alone does not qualify as an event service because of limitations in its addressing. The main problem is mapping expressions of interest into IP group addresses in a scalable way. Even assuming that a separate service, perhaps similar to DNS, is available for managing and resolving the mapping, the addressing scheme itself still poses major limitations in combining filters and patterns. Because MBone never relates two different IP groups, it would not be possible to exploit the compatibility relations between filters and thus it would not be possible to assemble filters into patterns. Notifications matching more than one filter or participating in more than one pattern would map into several multicast addresses, each one being routed in parallel and autonomously by MBone, thus defeating the whole purpose of the event service.

Event Notification Technologies

Some technologies specifically realize an event notification service, and some of them also attempt to extend their support to wide-area networks. To relate these systems to Siena, we adopt the classification framework defined in [2] and concentrate in particular on subscription languages.

Channel-based Subscriptions

The simplest subscription mechanism is the *channel*. Interested parties can listen to a channel by subscribing to it. An object of interest publishes a notification by addressing it to a specific channel; as a consequence, the notification is delivered to all the parties that are listening to that channel. Channel-based event services offer coarse-grained filtering and no patterns. Since channels define a partitioned address space for notifications, their service is equivalent to a reliable multicast communication. The CORBA Event Service [12] adopts a channel based architecture.

Subject-based Subscription

Some systems extend the concept of channel to *subject-based* addressing. In this case, event notifications contain a well-known attribute (the *subject*) that determines their address, while the remaining part of the notification is opaque for the event service. The main difference with respect to channels is that here subscriptions can express interest in many (potentially infinitely many) subjects/channels by specifying some form of expressions to match a subject. Also, in this model, two different subscriptions can define overlapping sets of notifications. TIBCO Rendezvous [8] adopts a subject-based subscription mechanism. In TIB Rendezvous, the subject is a list of strings over which it is possible to specify filters based on a limited form of regular expressions; for example, the filter `economy.exchange.*.MSFT*` will select all the notifications whose subject contains `economy` in its first position followed by `exchange` in second position, any string in third position, and a fourth string that starts with `MSFT`.

Content-based Subscription

By extending the domain of filters to the whole content of notifications we obtain another class of subscriptions called *content-based*. Content-based subscriptions are conceptually very similar to subject-based ones. However, by making the whole structured content of notifications visible to subscriptions, they give more freedom in the encoding the data upon which filters can be applied and which the event service can use for setting up routing information. Moreover, exposing the structure of notifications makes their type system (if any is adopted) visible too, thus, allowing more expressive and clear filters. Examples of systems that provide this kind of subscription language are JEDI [2], Yeast [10], GEM [11], Elvin [14], and Siena itself.

6 EXPERIENCE

The evaluation of distributed software systems is a difficult task, and the Siena event service is no exception. Because of its highly distributed nature, it is impossible to implement an event service and deploy it on a significant number of nodes just to see how it works. Not only it would be difficult to measure its performance, but also the cost of refining its topologies and algorithms would be too high at least in the early phases of its design.

So, in order to obtain feedback early in the design and development of Siena, and to have a quantitative evaluation of its topologies and algorithm, we adopted an approach that is common practice in the computer networks community for the validation of communication protocols and distributed systems in general. We chose to perform systematic simulations of different combinations of server topologies and dispatching algorithms in several network scenarios.

Simulation Framework

In simulating a network scenario, several models must be incorporated into the scenario. These models define the network topology, the layout of event servers, the population of applications (i.e., the distribution of interested parties and objects of interest), their behavior, etc. Clearly, every model makes certain simplifying assumptions. Also, every model is characterized by a significant number of parameters, which must be adjusted to reflect reality as faithfully as possible. These are the most important models we adopted in

our simulation framework together with their primary parameters:

- *network model*: The network model describes the physical (wide-area) communication network underlying the event service. The network is characterized by a graph. Each node in the graph represents a host or a cluster of nodes connected by a fast local-area subnet, and every edge represents a link with its latency and bandwidth. One way of modeling networks is to use a real network as a *benchmark* for which these parameters can be measured. The other way, adopted in our framework, is to use randomly generated graphs that are good approximations of the real network [15].
- *server model*: We use a layout of servers in which every network node hosts one server. We also assume that the connections between servers match the physical topology of the network. This second principle assumes that system administrators have a view of the topology of the immediate neighborhood of their subnet and that they can configure the event servers accordingly.
- *object distribution*: This model defines the distribution of objects of interest and interested parties among subnets. For now, we use an homogeneous distribution of objects. Thus, two parameters are given for the number per node of objects of interest and interested parties respectively.
- *objects behavior*: Although we could simulate real applications, we model applications as Poisson processes with respect to generation and consumption of events. Thus, the parameters that govern their behavior are the average time between two requests and the ratio between the number of requests issued per request type. This defines, for example, how many notifications are published for each advertisement.
- *computation and communication model*: Objects communicate by exchanging messages. These messages can carry event service requests (notifications, subscriptions, etc.) as well as control messages or forwarded messages that follow from some service request. Objects execute their own algorithm in response to messages or to the expiration of a time-out. Objects can send messages, set time-outs, create other objects or destroy other objects.

Running Simulations

Once a network scenario is defined, we run several simulations and collect traces of all the low-level messages exchanged between processes, hosts, and subnets. Then we analyze these traces by grouping messages according to some specific criteria (e.g., per host, per event service request, per type of request) and by computing the message count, minimum and maximum values, average value, and standard deviation of metrics such as network cost and delay. The network *cost* is a per-link parameter provided by the network model that accounts for the usage of communication resources in sending a message through a link. In our framework, we assume that the cost is proportional to the inverse of the bandwidth. By varying some parameters of the network scenario, such as the number of interested parties per node, we can plot these metrics and obtain indications of the behavior of that algorithm.

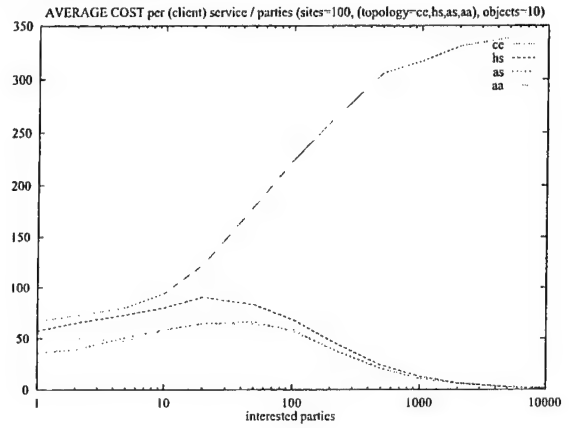


Figure 11: Scalability of some topologies and algorithms

Figure 11 shows the scalability of four combination of server topology and routing algorithm: ce = centralized, hs = hierarchical + subscription forwarding, as = acyclic + subscription forwarding, and aa = acyclic + advertisement forwarding. All the scenarios were executed on a network of a hundred nodes. The metric plotted is the average cost of messages grouped on a per-request basis; the unit of measure is not really relevant since we just want to compare different curves. The independent variable is the total number of interested parties.

The simulations that we performed so far clearly show that our topologies for distributed event services provide the scalability that can not be achieved with a centralized solution. In distinguishing the various distributed topologies, simulations show that the peer-to-peer topologies distribute the traffic evenly among servers, as opposed to the hierarchical topology that tends to over-load only a few nodes (at the highest level of the hierarchy). The simulator has been also a very effective testing tool for the development of the

routing algorithms.

7 CONCLUSION

This paper has described our work on Siena, a distributed, Internet-scale event notification service. We have described the design of the interface to the service, the algorithms and topologies we have designed to support event notification, and our ongoing simulation and evaluation work.

The simulation framework that we constructed has helped us significantly in refining topologies and algorithms. Also, the simulations confirm our intuitions about the scalability of the topologies and algorithms that we propose. However, we do not consider our evaluation effort to be complete. In fact, we plan on continuing our evaluation effort by exploring the parameter space in several directions. In particular, we are simulating different ranges of behavioral parameters to see which algorithms are most sensitive to different classes of applications.

We have implemented a prototype of Siena that realizes the acyclic topology with the subscription forwarding algorithm. We used this prototype as the event service of an agent-based deployment system called the SoftwareDock [7]. The current version of the prototype provides a reduced version of the notification model with only strings and integers and a few operators. Siena uses standard Internet technology, so its data model is transmitted in XML format, and servers are able to use straight TCP/IP as well as SMTP as a transport layer for messages. We plan on extending the prototype to implement the advertisement forwarding algorithm with a larger variety of types and operators and other transport layers including HTTP. This new version of the prototype will also allow us to apply the pattern matching optimizations that we discussed.

ACKNOWLEDGMENTS

We would like to thank Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, and André van der Hoek for their considerable contributions in discussing and shaping many of the ideas presented in this paper.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061, F30602-94-C-0253 and F30602-98-2-0163; and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] T. Barners-Lee. Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. Internet Request For Comments (RFC) 1630, Internet Engineering Task Force, June 1994.

[2] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. Technical report, CEFRIEL – Politecnico di Milano, Milano, Italy, August 1998. submitted for publication.

[3] Y. K. Dalal and R. M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, December 1978.

[4] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for Wide-Area Multicast Routing. *ACM Transactions on Networks*, April 1996.

[5] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Networks and Extended LANS. *ACM Transactions on Computer Systems*, 8(2):85–111, May 1990.

[6] W. Fenner. Internet Group Management Protocol, Version 2. Internet Request For Comments (RFC) 2236, Internet Engineering Task Force, November 1997.

[7] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, USA, May 1997.

[8] T. Inc. Rendezvous information bus. Technical report, TIBCO Inc., 1996. http://www.rv.tibco.com/rvwhitepaper.html.

[9] B. Kantor and P. Lapsley. Network news transfer protocol, a proposed standard for the stream-based transmission of news. Internet Request For Comments (RFC) 977, Internet Engineering Task Force, February 1986.

[10] B. Krishnamurthy and D. S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[11] M. Mansouri-Samani and M. Sloman. GEM A Generalized Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[12] Object Management Group. CORBAservices: Common Object Service Specification. Technical report, Object Management Group, July 1998.

[13] D. S. Rosenblum and A. L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the Sixth European Software Engineering Conference*, Zurich, Switzerland, September 1997. Springer-Verlag.

[14] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quencing. In *Proceedings of AUUG97*, July 1998.

[15] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom*, April 1996.

from the Proceedings of the 1998 International Conference on Software Engineering, Kyoto, Japan, April 18-25, 1998

Integrating Architecture Description Languages with a Standard Design Method

Jason E. Robbins **Nenad Medvidovic** **David F. Redmiles** **David S. Rosenblum**

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92697
{jrobbins,neno,redmiles,dsr}@ics.uci.edu

ABSTRACT

Software architecture descriptions are high-level models of software systems. Some researchers have proposed special-purpose architectural notations that have a great deal of expressive power but are not well integrated with common development methods. Others have used mainstream development methods that are accessible to developers, but lack semantics needed for extensive analysis. We describe an approach to combining the advantages of these two ways of modeling architectures. We present two examples of extending UML, an emerging standard design notation, for use with two architecture description languages, C2 and Wright. Our approach suggests a practical strategy for bringing architectural modeling into wider use, namely by incorporating substantial elements of architectural models into a standard design method.

Keywords

Software architecture, object-oriented design, architecture description languages, constraint languages, incremental development

1 INTRODUCTION

Architecture-based software development is an approach to designing software in which developers focus on one or more high-level models of the software system rather than program source code. Architectural models include elements such as software components, communication mechanisms, states, processes, threads, hosts, events, external systems, and source code modules [6, 9, 10, 17, 23, 24]. Relationships between these elements address such issues as message passing, data flow, resource usage, dependencies, state transitions, causality, and temporal orderings. The basic promise of software architecture research is that better software systems can be achieved by modeling their important aspects during development. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research [13].

Part of the software architecture research community, primarily academics, has focused on analytic evaluation of architecture descriptions. Answering difficult evaluation questions demands powerful modeling and analysis techniques that address specific aspects in depth. By paying the cost of making a detailed model, developers gain the benefit of knowing the answers to these questions. In this sense, software architecture descriptions serve primarily as input to analysis tools. For example, determining the possibility of deadlock requires specialized, formal models of the possible behavior and communication of each thread of control [3]. However, the emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formal methods draws the modeler's attention away from day-to-day development concerns. The use of special-purpose modeling languages has made this part of the architecture community fairly fragmented, as revealed by a recent survey of architecture description languages [14].

Another part of the community, primarily from industry, has focused on choosing which aspects to model. Modeling the wide range of issues that arise in software development demands a family of models that span and relate the issues of concern. By paying the cost of making such models, developers gain the benefit of clarifying and communicating their understanding of the system. In this sense software architectures serve primarily as the "big picture" of the system under development. For example, upgrading a database application requires an understanding of the various kinds of users and their respective tasks, the data schema, and the application's software components and their interfaces. However, emphasizing breadth over depth potentially allows many problems and errors to go undetected, because lack of rigor allows developers to ignore certain details. Several competing notations have been used in this part of the community, but they share central concepts, have been tempered by mainstream use, and have been formalized to some extent [4, 25]. There now exists a concerted effort to standardize methods for object-oriented analysis and design [16].

Standardization provides an economy of scale that results in more and better tools, better interoperability between tools, more available developers who are skilled in using that notation, and lower overall training costs. When special-purpose notations are needed, they can often be based on, or related to, standard notations. Doing so provides them with some of the benefits of the standard, and allows for more direct comparison and evaluation in terms of the value added by the special-purpose notation.

We use the Unified Modeling Language (UML) [18] as a starting point for bringing architectural modeling into wider use. UML is well suited for this because it provides a useful and extensible set of predefined constructs, it is semi-formally defined, it has substantial tool support, and it is based on experience with mainstream development methods. The next

section describes UML and our strategy for adapting it to our needs. Sections 3 and 4 provide examples of adapting UML with semantics specific to two ADLs, C2 and Wright. Section 5 expands on our approach and contrasts it to related work. Section 6 discusses the contributions of our approach: specifically, it is a way to integrate the power of ADLs with the day-to-day usefulness of UML; and more generally, it suggests a practical strategy for achieving partial integration of architectural models as needed for specific development tasks.

2 UML AND ITS EXTENSION MECHANISMS

2.1 UML Background

A UML model of a software system consists of several partial models, each of which addresses a certain set of issues at a certain level of fidelity. There are eight issues addressed by UML models: (1) classes and their declared attributes, operations, and relationships; (2) the possible states and behavior of individual classes; (3) packages of classes and their dependencies; (4) example scenarios of system usage including kinds of users and relationships between user tasks; (5) the behavior of the overall system in the context of a usage scenario; (6) examples of object instances with actual attributes and relationships in the context of a scenario; (7) examples of the actual behavior of interacting instances in the context of a scenario; and (8) the deployment and communication of software components on distributed hosts. Fidelity refers to how close the model will be to the eventual implementation of the system: low-fidelity models tend to be used early in the life-cycle and be more problem-oriented and generic, whereas high-fidelity models tend to be used later and be more solution-oriented and specific. Increasing fidelity demands effort and knowledge to build more detailed models, but results in more properties of the model holding true in the system.

The UML is a graphical language with well-defined syntax and semantics. The syntax of the graphical presentation is specified by examples and a mapping from graphical elements to elements of the underlying semantic model [20]. The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints [19]. The meta-model is itself a UML model that specifies the abstract syntax of UML models. This is much like using a BNF grammar to specify the syntax of a programming language. For example, the UML meta-model states that a Class is one kind of model element with certain attributes, and that a Feature is another kind of model element with its own attributes, and that there is a one-to-many composition relationship between them. Semantic constraints are expressed in the Object Constraint Language (OCL) which is based on first-order predicate logic [21]. Each OCL expression is evaluated in the context of some model element (referred to as "self") and may use attributes and relationships of that element as terms. OCL also defines common operations on sets and bags, and constructs for traversing relationships so that attributes of other model elements may also be used as terms. Traversing a one-to-many or many-to-many relationship results in a set of instances. Several examples of OCL constraints are given below.

2.2 UML Extension Mechanisms

UML is an extensible language in that new constructs may be added to address new issues in software development. Three mechanisms are provided to allow limited extension to new issues without changing the existing syntax or semantics of the language. (1) *Constraints* place semantic restrictions on particular design elements. (2) *Tagged values* allow new attributes to be added to particular elements of the model. (3) *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements.

Figure 1 presents an example of using UML to model part of a human resources system. A company employs many workers, offers many training courses, and owns many robots. Robots and employees are workers. Labor union contracts constrain companies such that robots may not make up more than 10% of the work force. A training course contains many trainees, and each trainee may take from 1 to 4 courses. In this example, Trainee is an interface (a set of operations) rather than a full class. An employee is capable of performing all the operations of Trainee.

Suppose we wish to impose the design constraint that "a person may not be a composite element of another class," in other words, "a person must be the whole in any whole-part relationships." This does not prevent a person from participating in containment relationships, only composite relationships. In UML, containment (white diamond) indicates that one object is temporarily subordinate to one or more others, whereas composition (black diamond) indicates that an object is subordinate to exactly one other object throughout its life-time. In this example, composition would mean that employees could not participate in any other aggregates and never work for another company. Constraints may be applied directly to a class or, as we have done here, constraints may be applied to a stereotype (e.g., Person) and the stereotype applied to a class (e.g., Employee). The constraint may be stated formally in OCL as:

Stereotype Person for instances of meta-class Class
[1] If a person is in any composite relationship, it must be the composite.

```
self.oclType.assocEnd.forAll(myEnd |
  myEnd.association.assocEnd->exists(anyEnd |
    anyEnd.aggregation = composite) implies
      myEnd.aggregation = composite)
```

Note: The above constraint is sufficient because the UML already constrains associations to have at most one composite end.

The labor union rule uses terms from the model to constrain the state of the system at run-time. In contrast, the Person stereotype uses terms from the UML meta-model to constrain the model of the system. Traversing the "oclType" association allows us to refer to the meta-model, rather than the design at hand. Figure 2 shows the parts of the UML meta-model used in this paper. We have simplified the meta-model for purposes of illustration, but all the constraints we define can be easily rewritten for use with the complete meta-model.

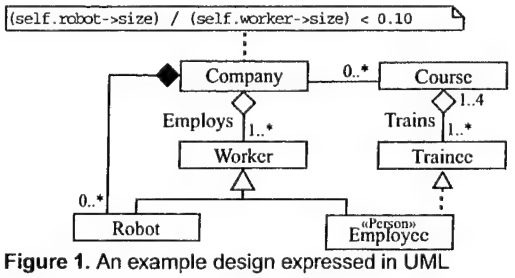
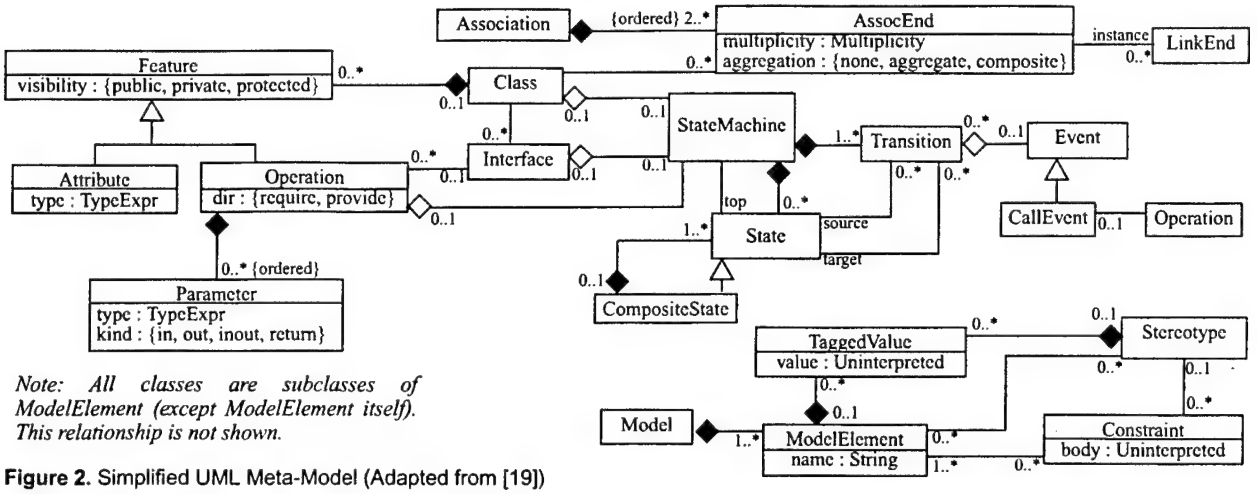


Figure 1. An example design expressed in UML



Note: All classes are subclasses of ModelElement (except ModelElement itself). This relationship is not shown.

Figure 2. Simplified UML Meta-Model (Adapted from [19])

2.3 Our Strategy for Adapting UML

One straightforward approach to using an ADL with UML is to define an ADL-specific meta-model. This approach has been used in more comprehensive formalization of architectural styles [1, 12]. Defining a new meta-model helps to formalize the ADL, but does not aid integration with standard design methods. By defining our new meta-classes as subclasses of existing meta-classes we would achieve some integration. For example, defining Component as a subclass of meta-class Class would give it the ability to participate in any relationship in which Class can participate. This is basically the integration that we desire. However, this integration approach requires *modifications* to the meta-model that would not *conform* to the UML standard, therefore we cannot expect UML-compliant tools to support it.

For the reason above, we restrict ourselves to using UML's built-in extension mechanisms on existing meta-classes. This allows the use of existing UML-compliant tools to represent the desired architectural models, and style conformance checking when OCL-compliant tools become available. Our basic strategy is to

- choose an existing meta-class from the UML meta-model that is semantically close to an ADL construct, then
- define a stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the ADL.

In the next two sections, we demonstrate this strategy and illustrate the results with example specifications.

3 INTEGRATING UML AND C2

3.1 Overview of C2

C2 is a software architecture style for user interface intensive systems [24]. C2 SADL is an ADL for describing C2-style architectures [12, 14]; henceforth we use "C2" to refer to the combination C2 and C2 SADL. In a C2-style architecture, *connectors* transmit messages between components, while *components* maintain state, perform operations, and exchange messages with other components via two interfaces (named "top" and "bottom"). Each interface consists of a set of messages that may be sent and a set of messages that may be received. Inter-component messages are either *requests* for a component to perform an operation, or *notifications* that a given component has performed an operation or changed state.

A C2 component consists of four internal parts. An *internal object* stores state and implements the operations that the component provides. A wrapper on the *internal object* monitors all requested operations and sends notifications through the bottom interface. A *dialog specification* maps from messages received to operations on the internal object. Optionally, a *translator* may modify some messages so as to match those understood by other components, thus adapting a component to fit into a particular architecture.

In the C2 style, components may not directly exchange messages; they may only do so via connectors. Each component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. Request messages may only be sent "upward" through the architecture, and notification messages may only be sent "downward."

The C2 style further demands that components communicate with each other only through message-passing, never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure *substrate independence*, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language in which the components or connectors are implemented, whether or not components have their own threads of control, the deployment of components to hosts, or the communication protocol used by connectors.

Figure 3 shows an example C2-style architecture. This system consists of four components and two connectors. One component is a database server, two are graphical user interfaces (GUI) to the database, and one is a window-system binding. One GUI is for posing queries, viewing result, and making updates. The other GUI is for configuring the database server. When either user interface is used to request a modification, a request message is sent upward to the connector, and then to the database. When the database

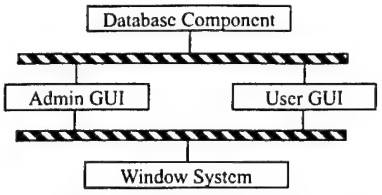


Figure 3. An example C2 architecture for a database application

performs an operation, a notification message is sent to the connector and is ultimately received by both GUI components. This style of component interaction is influenced by Model-View-Controller designs and supports multi-user systems and multi-view interfaces [8].

UML provides constructs for modeling software components, their interfaces, and their deployment on hosts. However, these built-in constructs are not suitable for describing C2-style software architectures because they assume both too much and too little. Components in UML are assumed to be concrete executable artifacts that take up machine resources such as memory. In contrast, C2 components are conceptual artifacts that decompose the system's state and behavior. C2 components may be implemented by concrete components, but they are not themselves concrete. Furthermore, components in UML may have any number of interfaces and any internal structure, whereas C2 components must follow the C2-style rules. Since "vanilla" UML does not fit our needs, we will adapt it to express several aspects of the C2 style.

3.2 C2 Operations in UML

The UML meta-class Operation matches the C2 concept of a message specification. UML Operations consist of a name and a parameter list (which may contain returned values). Operations indicate whether they will be provided or required (i.e., they may be received or sent). Operations may be public, private, or protected. To model C2 message specifications we add a tag to differentiate notifications from requests and constrain Operation to have no return values. C2 messages are all public, but that constraint is built into the UML meta-class Interface used below.

Stereotype C2Operation for instances of meta-class Operation
[1] C2Operations are tagged as either notifications or requests.
```
c2MsgType : enum { notification, request }
```

[2] C2 messages do not have return values.
```
self.parameter->forAll(p | p.kind <> return)
```

3.3 C2 Components in UML

The UML meta-class Class is closest to C2's notion of component. Classes may provide multiple interfaces with operations, may own internal parts, and may participate in associations with other classes. However, there are aspects of Class that are not appropriate, namely, they may have methods and attributes. In UML, an operation is a specification of a procedural abstraction (i.e., a procedure signature with optional pre- and post-conditions), while a method is a procedure body. Components in C2 provide only operations, not methods, and those operations must be part of interfaces provided by the component, not directly part of the component. Furthermore, a C2 conceptual component is assumed to have no state other than the state of its internal parts, and thus may have no direct attributes.

Stereotype C2Interface for instances of meta-class Interface
[1] A C2 interface has a tagged value identifying its position.
```
c2pos : enum { top, bottom }
```

[2] All C2Interface operations must have stereotype C2Operation.
```
self.oclType.operation->forAll(o | o.stereotype = C2Operation)
```

Stereotype C2Component for instances of meta-class Class
[1] C2Components may not directly contain features (i.e., methods, operations, or attributes).
```
self.oclType.feature->size = 0
```

[2] C2Components must implement exactly two interfaces, which must be C2Interfaces, one top, and the other bottom.
```
self.oclType.interface->size = 2 and
self.oclType.interface->forAll(i |
  i.stereotype = C2Interface) and
self.oclType.interface->exists(i | i.c2pos = top) and
self.oclType.interface->exists(i | i.c2pos = bottom)
```

[3] Requests travel "upward" only, i.e., they are sent through top interfaces and received through bottom interfaces.
```
Let topInt = self.oclType.interface->select(i |
    i.c2pos = top),
Let botInt = self.oclType.interface->select(i |
    i.c2pos = bottom),
topInt.operation->forAll(o |
  (o.c2MsgType = request) implies o.dir = require) and
botInt.operation->forAll(o |
  (o.c2MsgType = request) implies o.dir = provide)
```

[4] Notifications travel "downward" only. Similar to the constraint above.

[5] Each C2Component has one instance in the running system.
```
self.allInstances->size = 1
```

[6] C2Components participate in at most four whole-part relationships named internalObject, wrapper, dialog, and translator.
```
Let wholes = self.oclType.assocEnd->select(
    aggregation = composite),
(whole->size <= 4) and
((wholes.association.name->asSet) - Set {
    "internalObject", "wrapper", "dialog",
    "translator"})->size = 0
```

[7] Each operation on the internal object has a corresponding notification which is sent from the component's bottom interface.
```
Let ops = self.internalObject.feature->select(f |
    f->isKindOf(Operation)),
Let botInt = self.oclType.interface->select(i |
    i.c2pos = bottom),
ops->forAll(op |
  botInt->exists(note |
    (op.name = note.name and
      op.parameter = note.parameter) implies
        note.dir = required and note.c2MsgType = notification))
```

3.4 C2 Connectors in UML

C2 connectors share many of the constraints of C2 components. One difference is that they do not have any prescribed internal structure. Components and connectors are treated differently in the architecture composition rules discussed below. Another difference is that connectors may not define their own interfaces; instead their interfaces are determined by the components that they connect.

We can model C2 connectors using a stereotype C2Connector that is similar to C2Component. Below, we reuse some constraints and add two new ones. But first, we introduce three stereotypes for modeling the attachments of components to connectors. These attachments are needed to determine component interfaces.

Stereotype C2AttachOverComp for instances of meta-class Association
[1] C2 attachments are binary associations.
```
self.oclType.assocEnd->size = 2
```

[2] The first end of the association must be to a C2 component.
```
Let ends = self.oclType.assocEnd,
ends[1].multiplicity = "1..1" and
ends[1].class.stereotype = C2Component
```

[3] The second end of the association must be to a C2 connector.
```
Let ends = self.oclType.assocEnd,
ends[2].multiplicity = "1..1" and
ends[2].class.stereotype = C2Connector
```

Stereotype C2AttachUnderComp for instances of meta-class Association. Same as C2AttachOverComp, except that the first end must be to a connector, and the second end must be to a component.

Stereotype C2AttachConnConn for instances of meta-class Association
[1] C2 attachments are binary associations.
```
self.oclType.assocEnd->size = 2
```

[2] Each end of the association must be on a C2 connector.
```
self.oclType.assocEnd->forAll(ae |
  ae.multiplicity = "1..1" and
  ae.class.stereotype = C2Connector)
```

[3] The two ends are not both on the same C2 connector.
```
self.oclType.assocEnd[1].class <>
      self.oclType.assocEnd[2].class
```

Stereotype C2Connector for instances of meta-class Class
[1-5] Same as constraints 1-5 on C2Component.

[6] The top interface of a connector is determined by the components and connectors attached to its bottom.
```
Let topInt = self.oclType.interface->select(i | i.c2pos = top),
Let downAttach = self.oclType.assocEnd.associa-
    tion->select(a | a.assocEnd[2] = self.oclType),
Let topsIntsBelow = downAttach.assocEnd[1].inter-
    face->select(i | i.c2pos = top),
topsIntsBelow.operation->asSet = topInt.operation->asSet
```

[7] The bottom interface of a connector is determined by the components and connectors attached to its top. This is similar to the constraint above.

3.5 C2 Architectures in UML

Now we turn our attention to the overall composition of components and connectors in the architecture of a system. Recall that well-formed C2 architectures consist of components and connectors, components may be attached to one connector on the top and one on the bottom, and the top (bottom) of a connector may be attached to any number of other connectors' bottoms (tops). Below, we also add two new rules that guard against degenerate cases.

Stereotype C2Architecture for instances of meta-class Model
[1] A C2 architecture is made up of only C2 model elements.
```
self.oclType.modelElement->forAll(me |
  me.stereotype = C2Component or
  me.stereotype = C2Connector or
  me.stereotype = C2AttachOverComp or
  me.stereotype = C2AttachUnderComp or
  me.stereotype = C2AttachConnConn)
```

[2] Each C2Component has at most one C2AttachOverComp.
```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps->forAll(c |
    c.assocEnd.association->select(a |
       a.stereotype = C2AttachUnderComp)->size <= 1)
```

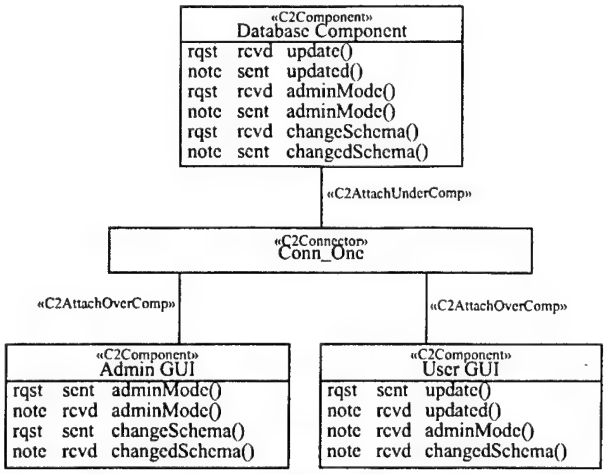


Figure 4. C2 architecture from Figure 3 expressed in UML

[3] Each C2Component has at most one C2AttachUnderComp. Similar to the constraint above.

[4] C2Components do not participate in any non-C2 associations.
```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps.assocEnd.association->forAll(a |
  a.stereotype = C2AttachOverComp or
  a.stereotype = C2AttachUnderComp)
```

[5] C2Connectors do not participate in any non-C2 associations.
```
Let conns = self.oclType.modelElement->select(me |
  me.stereotype = C2Connector),
conns.assocEnd.association->forAll(a |
  a.stereotype = C2AttachOverComp or
  a.stereotype = C2AttachUnderComp or
  a.stereotype = C2AttachConnConn)
```

[6] Each C2Component must be attached to some connector.
```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps->forAll(c |
    c.assocEnd.association->size > 0)
```

[7] Each C2Connector must be attached to some connector or component.
```
Let conns = self.oclType.elements->select(e |
    e.stereotype = C2Connector),
conns->forAll(c |
    c.assocEnd.association->size > 0)
```

3.6 Example C2 Architecture

Figure 4 shows the UML graphical notation for the same system shown in Figure 3 to illustrate the C2 style. We show some operations and omit others as needed to clarify the discussion below. Each element is marked with its stereotype in small double angle brackets. Alternatively, UML allows icons to be used to denote the stereotype.

Given a C2 architecture that is modeled in UML, it can be related to other standard UML model elements that are commonly used in software development. Figure 5 makes explicit our assumptions about the kinds of users who will use this system and their tasks. Figure 6 is a sequence diagram showing how the system behaves in the context of a particular use case. Explicitly modeling these aspects of the system enhances C2's support for component-based development of systems with complex user interfaces.

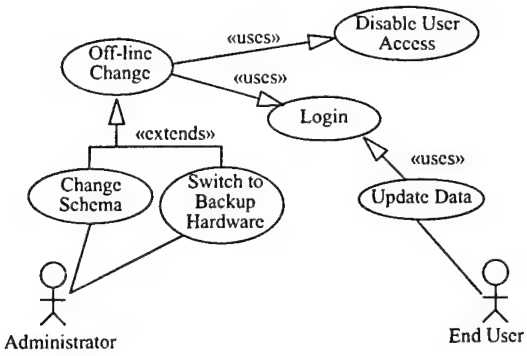


Figure 5. Some use cases for the example database system

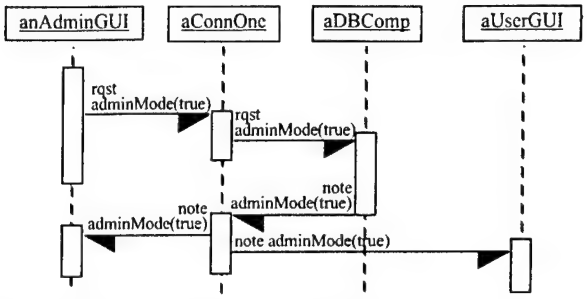


Figure 6. Sequence diagram for Disable User Access

3.7 Benefits of Integrating UML and C2

Adapting UML to enforce the C2-style rules has been fairly straightforward, because many C2 concepts are found in UML. Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language. Neither C2 nor UML constrain the choice of GUI toolkits or inter-process communication mechanisms. Neither C2 nor UML (as we have used it) assume that any two components run in the same thread of control or on the same host. Both C2 and UML limit communication to message passing and include specifications of messages that may be sent and received. Although we did not model details of the internal parts of a C2 component or the behavior of any C2 constructs, we feel those aspects of C2 could be modeled in UML. In fact, we provide an example of modeling behavior in the next section.

Some concepts of C2 are very different from those of UML and object-oriented design in general. For example, mainstream object-oriented design has a strict dichotomy between classes and instances. Since each class may have multiple instances, associations between classes may have multiplicity greater than one (e.g., there could be any number of employees in Figure 1). Furthermore, the features of an instance are declared in its class. In contrast, the interface of a C2 connector is determined by context rather than declared, and the addition of a new component instance at run-time is considered an architectural change. We addressed this difference by demanding that each C2 component and connector have exactly one instance. If a system uses two connectors, they must each have their own class in the design, although they may be implemented by the same concrete components. Another conceptual difference is that it is legal for C2 messages to be sent and not received by any component, whereas UML assumes that every message sent will be received. We have declined to address this last difference since it introduces more complexity than we feel it merits. As will be discussed more in Section 5, our approach allows aspects of an ADL to be expressed in UML or left to special purpose tools as desired.

4 INTEGRATING UML AND WRIGHT

The preceding section demonstrated that an ADL that supports a specific architectural style can be modeled in UML. This section shows the applicability of our approach to a general-purpose ADL, Wright [2, 3]. A more recent version of Wright also supports system families, architectural styles, and hierarchical composition. We do not address these newer features here, but believe that they could be incorporated into our model.

An architecture in Wright is described in three parts:

- component and connector types;
- component and connector instances; and
- configurations of component and connector instances.

Unlike C2, Wright does not enforce the rules of a particular style, but is applicable to multiple styles. However, it still places certain topological constraints on architectures. For example, as in C2, two components cannot be directly connected, but must communicate through a connector; on the other hand, unlike C2, Wright disallows two connectors from being directly attached to one another.

The remainder of the section describes an extension to UML for modeling Wright architectures. For brevity, stereotypes and constraints are elided whenever they are obvious from the discussion in this or the previous section.

4.1 Behavioral Specification in Wright

Wright uses a subset of CSP [7] to provide a formal basis for specifying the behavior of components and connectors, as well as the protocols supported by their interface elements. Given that this subset "defines processes that are essentially finite state" [2], it is possible to model Wright's behavioral specifications using UML's State Machines [20].

CSP processes are entities that engage in communication events. An event, e, can be primitive, or it can input or output a data item x (denoted in CSP with `e?x` or `e!x`, respectively). CSP events are modeled in State Machines as shown in Figure 7.

These two types of state transitions can be used in modeling more complex CSP expressions supported by Wright. Table 1 presents the mapping from CSP to State Machines using events with no actions (Figure 7a); the mapping for null events with actions (Figure 7b) is straightforward. It is possible for CSP events to have no associated data (see Figure 8 below). In such a case, the semantics of State Machines force us to make a choice as to which entities generate events and which observe them. We choose to model Wright ports and roles (described below) with event-generating actions, and computation and glue with transitions that observe those events.

The state machines in Table 1 can be used as templates from which equivalents of more complex CSP expressions can be

Figure 7. (a) A CSP event with input data, e?x, is modeled in UML State Machines as a state transition event with no action. (b) A CSP event, e, with output data, e!x, is modeled as a null state transition event that results in action e.

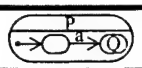
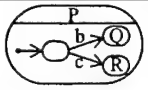
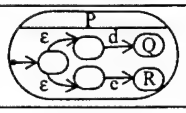
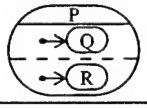
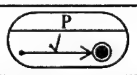
CSP Concept	CSP Notation	UML State Machine
Prefixing	$P = a \rightarrow Q$	
Alternative (deterministic choice)	$P = b \rightarrow Q \Box c \rightarrow R$	
Decision (non-deterministic choice)	$P = d \rightarrow Q \sqcap e \rightarrow R$	
Parallel Composition	$P = Q \parallel R$	
Success Event	$P = \surd$	

Table 1. UML State Machine templates for Wright's CSP constructs

formed. Therefore, a "Wright" state machine is described by the following stereotypes.

Stereotype WSMTransition for instances of meta-class Transition
[1] A transition is tagged as one of the two cases shown in Figure 7.
```
WSMtransitionType : enum { event, action }
```

[2] An "event" transition consists of an event only (Figure 7a).
```
self.oclType.WSMtransitionType = event implies
  (self.oclType.event.oclIsTypeOf(CallEvent) and
  self.oclType.ActionSequence->size = 0)
```

[3] An "action" transition consists of a null event and an action (Figure 7b).
```
self.oclType.WSMtransitionType = action implies
  (self.oclType.event->size = 0 and
  self.oclType.ActionSequence.Action->size = 1)
```

Stereotype WrightStateMachine for instances of metaclass StateMachine
[1] A WrightStateMachine consists of one of the composite states discussed above, and partially depicted in Table 1. Each simple state may be refined as another WrightStateMachine. This constraint is elided in the interest of space.

[2] All WrightStateMachine transitions must be WSMTransitions.
```
self.oclType.transition->forAll(t | t = WSMTransition)
```

4.2 Wright Component and Connector Interfaces in UML

Each Wright interface (a *port* in a component or a *role* in a connector) has one or more operations. In Wright, these operations are modeled implicitly, as part of a port or role's CSP protocol. We choose to model the operations explicitly in UML. The CSP protocols associated with a port or role are modeled as WrightStateMachines.

Stereotype WrightOperation for instances of meta-class Operation
[1] WrightOperations do not have parameters; parameters are implicit in the CSP specification associated with each operation
```
self.parameter->size = 0
```

Stereotype WrightInterface for instances of meta-class Interface
[1] WrightInterfaces are tagged as either ports or roles.
```
WrightInterfaceType : enum { port, role }
```

[2] All operations in a WrightInterface are WrightOperations.
```
self.oclType.operation->forAll(o |
  o.stereotype = WrightOperation)
```

[3] Exactly one WrightStateMachine is associated with each WrightInterface.
```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(s |
  s.stereotype = WrightStateMachine)
```

[4] In a WrightInterface, a WrightStateMachine is expressed only in terms of that interface's operations; these are operations on the state machine's call events.
```
self.oclType.stateMachine.transition->forAll(t |
  (t.event.oclIsTypeOf(CallEvent)) implies
    self.oclType.operation->exists(o |
      o = t.event.operation))
```

A WrightInterface, as modeled above, specifies the alphabet of a port or role.

4.3 Wright Connectors in UML

A connector type in Wright is described as a set of *roles*, which describe the expected behavior of the interacting components, and a *glue*, which defines the connector's behavior, by specifying how its roles interact.

We will model Wright connectors with the UML meta-class Class. Wright connectors provide multiple interfaces (roles) and participate in associations with other classes (Wright components). Wright connector types are assumed to have no state other than the state of their internal parts, and thus may have no direct attributes.

Stereotype WrightGlue for instances of meta-class Operation
[1] WrightGlue is modeled as a WrightOperation.
```
self.oclType.operation->forAll(o |
  o.stereotype = WrightOperation)
```

[2] WrightGlue contains a single WrightStateMachine.
```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(s |
  s.stereotype = WrightStateMachine)
```

Stereotype WrightConnector for instances of meta-class Class
[1] WrightConnectors must implement at least one WrightInterfaceType, which must be a role.
```
self.oclType.interface->size >= 1 and
self.oclType.interface->forAll(i |
  i.stereotype = WrightInterface and
  i.WrightInterfaceType = role)
```

[2] A WrightConnector contains a single glue.
```
self.oclType.operation->size = 1 and
self.oclType.operation->forAll(o |
  o.stereotype = WrightGlue)
```

[3] Operations with no data and with input data that belong to the different interface elements of a connector are the trigger events in glue's state machine.
```
self.oclType.operation.stateMachine.transition->forAll(t |
  (t.event.oclIsTypeOf(CallEvent)) implies
    self.oclType.interface.operation->exists(o |
      o = t.event.operation))
```

[4] Operations with output data that belong to the different interface elements of a connector are the actions in glue's state machine. Similar to the above constraint.

[5] The semantics of a Wright connector can be described as the parallel interaction of its glue and roles [2].

```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(sm |
  sm.state->size = 1 and sm.state->forAll(s |
    s.oclType = CompositeState and s.isConcurrent = true and
    s.state->size = 1 + self.oclType.interface->size and
    s.state->exists (gs |
      gs = self.oclType.operation.stateMachine.top) and
    self.oclType.interface->forAll(i |
      s.state->exists (rs | rs = i.stateMachine.top))))
```

[6] A WrightConnector must have at least one instance in the running system.

```
self.allInstances->size >= 1
```

4.4 Wright Components in UML

A component type is modeled by a set of *ports*, which export the component's interface, and a *computation* specification, which defines the component's behavior. We model Wright components in UML with a stereotype WrightComponent. This stereotype has much in common with the WrightConnector stereotype, and is thus omitted.

4.5 Wright Architectures in UML

We introduce stereotypes for modeling the attachments of components to connectors and for Wright architectures. Unlike C2, which considers architectures to be networks of abstract placeholders, Wright architectures are composed of component and connector instances. One solution we considered was to define WrightConnectorInstance and WrightComponentInstance stereotypes and express architectural topology in terms of them. However, we believe that it is undesirable to introduce instances at this level, since we are still dealing with design issues. Additionally, we have found that most of the constraints on component and connector instances can be expressed in terms of their corresponding types. Therefore, we refer to component and connector types in the stereotypes below.[1]

Stereotype WrightAttachment for instances of meta-class Association

[1] Wright attachments are associations between two elements.

```
self.oclType.assocEnd->size = 2
```

[2] One end of the association must be to a Wright component.

```
Let ends = self.oclType.assocEnd,
ends[1].multiplicity = "1..1" and
ends[1].class.stereotype = WrightComponent
```

[3] The other end of the association must be to a Wright connector.

```
Let roles = self.oclType.assocEnd,
ends[2].multiplicity = "1..1" and
ends[2].class.stereotype = WrightConnector
```

Stereotype WrightArchitecture for instances of meta-class Model

[1] A WrightArchitecture is made up of only Wright model elements.

```
self.oclType.elements->forAll(e |
  e.stereotype = WrightComponent or
  e.stereotype = WrightConnector or
  e.stereotype = WrightAttachment)
```

[2] Each WrightComponent port participates in at most one WrightConnector role.

```
Let comps = self.oclType.elements->select(e |
  e.stereotype = WrightComponent),
comps.assocEnd->forAll(ae | ae.linkEnd->size = 1)
```

[3] Each WrightConnector role is fulfilled by at most one WrightComponent port. Similar to the constraint above.

[4] WrightComponents and WrightConnectors do not participate in any non-Wright associations. Similar to constraints [4-5] in Section 3.5.

1. The one exception is in constraints 2 and 3 of the WrightArchitecture stereotype: "linkEnd" refers to an instance of a class (type).

```
connector Pipe =
  role Writer = write → Writer ⊓ close → √
  role Reader =
    let ExitOnly = close → √
    in let DoRead = (read → Reader
                  □ read-eof → ExitOnly)
    in DoRead ⊓ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly
                      □ Reader.read-eof
                          → Reader.close → √
                      □ Reader.close → √
         in let WriteOnly = Writer.write → WriteOnly
                          □ Writer.close → √
         in Writer.write → glue
           □ Reader.read → glue
           □ Writer.close → ReadOnly
           □ Reader.close → WriteOnly
```

Figure 8. A connector specified in Wright (adapted from [2])

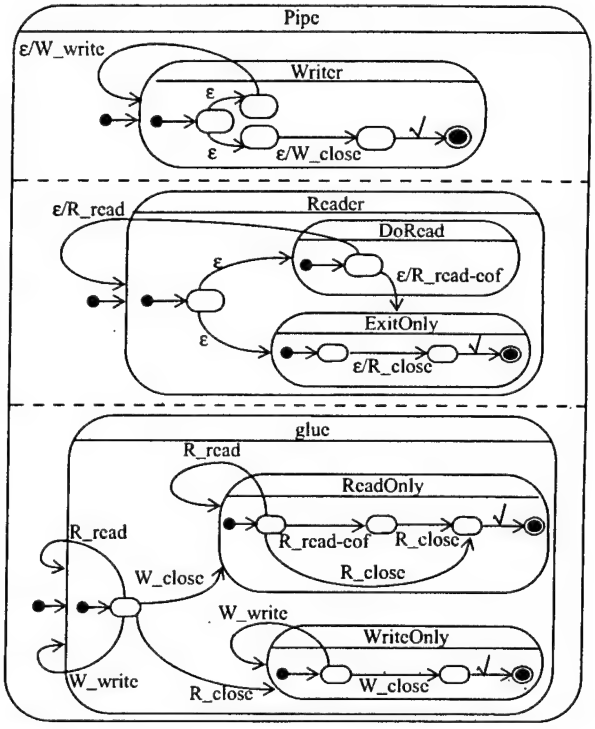


Figure 9. UML State Machine model of the *Pipe* connector

The semantics of port-role attachments in Wright are formally defined [3]. However, Wright places no language-level constraints on port-role pairs. Instead, establishing and enforcing these constraints is the task of external analysis tools. Hence, we provide no port-role compatibility constraints.

4.6 Example Partial Wright Architecture

Having provided an extension to UML for modeling Wright architectures, we now demonstrate how that extension is used to describe a Wright specification. Figure 8 shows the *Pipe* connector example from [2]. The UML State Machine model of the *Pipe* is shown in Figure 9. Wright's scoping of events is modeled in UML by prefixing every event's name with the name of the role to which the event belongs. The

class diagram for *Pipe* is analogous to the C2 diagram shown in Figure 4, and has been omitted for brevity

4.7 Benefits of Integrating UML and Wright

Modeling an ADL such as Wright in UML provides benefits both to practitioners who prefer Wright as a design notation and to those who are more familiar with UML. Mapping a Wright architecture to UML enables a Wright user to leverage a wide number of general-purpose UML tools (e.g., code generation, simulation, analysis, reverse engineering, and so forth). On the other hand, being able to map a UML design of a system to Wright (by adhering to the constraints specified in this section) would enable a UML designer to utilize Wright's powerful analysis capabilities, such as interface compatibility checking and deadlock detection.

5 CORE MODELS AND EXTENSIONS

Notational standardization has a wide range of benefits, as discussed in the introduction. The challenge of standardization is finding a language that is general enough to capture needed concepts without adding too much complexity. It is tempting to extend the UML meta-model to fully capture each feature of each ADL. However, such a notation would be overly complex and incompatible with standard UML tools. There has never been a single programming language that served the needs of all programmers, and there is no reason to expect a single ADL to meet the needs of all software architects. This has led the software architecture community to attempt interchange rather than standardization of ADLs.

ACME is an architecture interchange language that supports automatic transformation of a system modeled in one ADL to an equivalent model in another ADL [5]. This allows architects to model and analyze their system architecture in one ADL and then translate the model to another ADL for further analysis. Architects need not work directly with ACME; they may instead use the ADL and toolset that is most suited to the current issue of concern. ACME's approach is easier than providing direct mappings between pairs of ADLs because the ACME language serves as an intermediate step and provides additional tool support. ACME's *architectural ontology* plays a role analogous to UML's meta-model; however it is smaller and focuses on structural aspects of architectures.

Full realization of ACME's goals presents a number of challenges. Complete, automated translation among a set of ADLs requires a set of semantic mappings that involve every concept of every ADL in the set, which may not be possible given that different ADLs address different system aspects and have different semantics. The translation approach depends on exploiting constructs common to every ADL. At this point, the evident commonalities are syntactic rather than semantic [14]. For these reasons ACME emphasizes a partial and incremental approach.

ACME uses a seven element architectural ontology together with key-value pairs to represent arbitrary, uninterpreted architectural features and a template mechanism that leverages commonalities. Like ACME, our approach uses a fixed ontology (the UML meta-model), key-value pairs (tagged-values), and templates (stereotypes). However, UML provides much richer semantics due to its more comprehensive meta-model and its first-order predicate logic constraints.

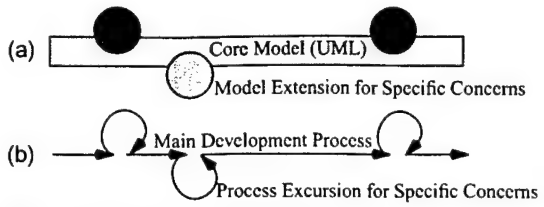


Figure 10. (a) A core model with extensions
(b) Sketch of an associated process

A fundamental difference is that our approach does not use translation between notations, but rather uses a core model with several independent extensions. We use UML as our core model and assume that developers are able to use UML constructs, such as classes and use cases, in day-to-day development activities. We extend this core model with specific attributes and constraints as needed for specific analyses. As new issues of concern arise in development, new attributes may be added to support analyses relevant to those concerns. The semantics of the core model are always enforced by UML-compliant tools. The semantics of each extension are enforced by the constraints of that extension and the constraints imposed by the desired analyses. Dependencies and conflicts may arise between the attributes in different extensions, and must be handled by developers just as they manage the other myriad dependencies and potential conflicts of software development. This situation is not ideal, but it is practical: it uses available methods and tools that are well integrated into day-to-day development, and it is incremental. We feel that these features are key to bringing the benefits of architectural modeling into mainstream use.

In using a core model and extensions, the question arises of what should be in the core and what should be left to extensions. Technical considerations play some role in this decision. For example, ACME's simple architectural ontology eases tool building, whereas UML's larger meta-model presents a higher barrier. Development processes also influence the core model. For example, object-oriented design and use cases are widely used by practitioners and directly relate to day-to-day development activities. We choose UML as our core model because it is grounded in mainstream development practices, already has substantial tool support, and provides explicit extension mechanisms.

Figure 10 sketches a process in which developers use the core model and some available extensions for day-to-day development concerns and take *process excursions* as needed to address specific architectural concerns identified during the main process. Information learned in excursions guides later decisions in the main process. Different concerns will arise as the main process progresses and model fidelity increases. For example, deadlock can only be addressed once system behavior is specified in detail. We envision developers using UML normally and ADL-specific tools as needed; an alternative process more suited to researchers might involve using an ADL normally and UML tools as needed (e.g., to generate code).

6 CONCLUSIONS

Further research into this approach will attempt to integrate UML with the semantics of other ADLs, apply object-oriented concepts such as polymorphism and inheritance to architectural elements [15], exploit more formal semantics [4, 25] and

evaluate the effectiveness of the approach in practice. In addition to C2 and Wright, we have also investigated integrating UML with Darwin [11] and Rapide [10]. Each of these ADLs has certain aspects in common with UML, some of which can be expressed with UML's extension mechanisms, while others may be included in a UML specification but can only be interpreted by ADL-specific tools.

From our experience to date, adapting UML to address architectural concerns seems to require reasonable effort, be a useful complement to ADLs and their analysis tools, and be a practical step toward mainstream architectural modeling. Using UML has the benefits of leveraging mainstream tools, skills, and processes. It may also aid in comparison of ADLs because it forces some implicit assumptions to be explicitly stated in common terms.

Integrating architectural models into mainstream development methods is not simply a matter of convenience. Based on experience in complex system design, "mismatches between the systems models used by the R&D design team and those of the system engineer, manufacturer, and user have delayed delivery, raised costs, entailed product rework, and led to faulty failure diagnoses [22]." These problems arise when models become out of synch with the system and current design concerns, or when lessons learned in modeling are not communicated to developers. Integrating architectural models with standard design methods addresses both these issues.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9624846 and Grant No. CCR-9701973. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-97-2-0021 and F30602-94-C-0218, and Air Force Office of Scientific Research under grant number F49620-98-1-0061. Additional support is provided by Rockwell International and Northrop Grumman Corp. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

REFERENCES

1. Abowd, G., Allen, R., and Garlan, D. Formalizing style to understand descriptions of software architecture. Trans. Software Engineering and Methodology. October 1995. pp. 319-364.
2. Allen, R. and Garlan, D. Formalizing Architectural Connection. Proceedings of the 1994 International Conference on Software Engineering, Sorrento, Italy, May 1994. pp. 71-80.
3. Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology. vol. 6. no. 3. July 1997. pp. 213-249.
4. Bourdeau, R. and Cheng, B. A formal semantics for object model diagrams. IEEE Trans. on Software Engineering. vol. 21. no. 10. October 1995. pp. 799-821.
5. Garlan, D., Monroe, R. and Wile D. ACME: An architectural interconnection language. Proc. of CASCON'97. Toronto, Canada. November 1997.
6. Garlan, D. and Shaw M. An introduction to software architecture: Advances in software engineering and knowledge engineering, volume I. World Scientific Publishing, 1993.
7. Hoare, C. A. R. Communicating Sequential Processes. Prentice Hall, 1985.
8. Krasner, G. E. and Pope, S. T. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. J. Object-Oriented Programming. vol. 1. no. 3, Aug/Sept 1988. pp. 26-49.
9. Kruchten, P. B. The 4+1 view model of architecture. IEEE Software. Nov. 1995. pp. 42-50.
10. Luckham, D. C., and Vera, J. An event-based architecture definition language. IEEE Transactions on Software Engineering. vol. 21. no. 9. September 1995. pp. 717-734.
11. Magee, J. and Kramer, J. Dynamic structures in software architecture. Proc. of SIGSOFT'96. San Francisco, CA, October 1996.
12. Medvidovic N., Taylor, R. N., Whitehead, Jr. E. J. Formal modeling of software architectures at multiple levels of abstraction. Proc. California Software Symposium, Los Angeles, April 1996. pp. 29-40.
13. Medvidovic, N. and Rosenblum, D. S. Domains of concern in software architectures and architecture description languages. Proc. USENIX Conf. on Domain Specific Languages, Santa Barbara, CA, October. 1997. pp. 199-212.
14. Medvidovic, N. and Taylor, R. N. A framework for classifying and comparing architecture description languages. The Sixth European Software Engineering Conference together with SIGSOFT'97. Zurich, Switzerland, September 1997. pp. 60-76.
15. Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. SIGSOFT'96. pp 24-32. San Francisco, CA, October 1996.
16. Object Management Group. Object analysis and design RFP-1. Object Management Group document ad/96-05-01. June 1996. Available from http://www.omg.org/docs/ad/96-05-01.pdf.
17. Perry, D. E. and Wolf, A. L. Foundations for the study of software architectures. Software Engineering Notes. October 1992.
18. Rational Partners (Rational, IBM, HP, Unisys, MCI, Microsoft, ObjecTime, Oracle, i-Logix, etc.). Proposal to the OMG in response to OA&D RFP-1. Object Management Group document ad/97-07-03. July 1997. Available from http://www.omg.org/docs/ad/.
19. Rational Partners. UML Semantics. Object Management Group document ad/97-08-04. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-04.pdf.
20. Rational Partners. UML Notation Guide. Object Management Group document ad/97-08-05. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-05.pdf.
21. Rational Software Corporation and IBM. Object constraint language specification. Object Management Group document ad/97-08-08. Sept. 1997. Available from http://www.omg.org/docs/ad/.
22. Rechtin, E. The synthesis of complex systems. IEEE Spectrum, July 1997. pp. 51-55.
23. Soni, D., Nord, R., and Hofmeister C. Software architecture in industrial applications. Proc. of the 17th International Conference on Software Engineering. Seattle, WA. 1995. pp. 196-207.
24. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component and message-based architectural style for GUI software. IEEE Trans. Software Engineering, June 1996, vol.22, no.6, pp.390-406.
25. Wang, E., Richter, H., and Cheng, B. Formalizing and integrating the dynamic model within OMT. Proc. IEEE International Conference on Software Engineering (ICSE'97). Boston, MA. May 1997. pp. 45-55.

Challenges in Exploiting Architectural Models for Software Testing

David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
+1 949.824.6534
dsr@ics.uci.edu
http://www.ics.uci.edu/~dsr/

ABSTRACT

Software architectural modeling offers a natural framework for designing and analyzing modern large-scale software systems and for composing systems from reusable off-the-shelf components. However, the nature of component-based software presents particularly unique challenges for testing component-based systems. To date there have been relatively few attempts to establish a sound theoretical basis for testing component-based software.

This paper discusses challenges in exploiting architectural models for software testing. The discussion is framed in terms of the author's recent work on defining a formal model of test adequacy for component-based software, and how this model can be enhanced to exploit formal architectural models.

Keywords

ADLs, architectural modeling, component-based software, integration testing, software testing, subdomain-based testing, test adequacy criterion, unit testing.

1 INTRODUCTION

Software architectural modeling offers a natural framework for designing and analyzing modern large-scale software systems and for composing systems from reusable off-the-shelf components [9,10]. However, the nature of component-based software presents particularly unique challenges for testing component-based systems. In particular, while the technology for constructing component-based software is relatively advanced, and while the architecture research community has produced a number of powerful formal notations and analysis techniques for architectural modeling, there have been relatively few attempts to establish a sound theoretical basis for testing component-based software (e.g., see [3,11,12]).

An architectural model can be used in a variety of ways to aid the testing of a component-based system:

- The model itself can be tested directly, prior to the selection of components and the development of the implementation. This requires that the model be expressed in an *architecture description language* (ADL) having a simulation or execution semantics [6]. Rapide is an example of such a language [4].
- The model can be used to guide integration testing of the implemented system. In particular, the structure of the model can be used to guide the order in which components are assembled and tested, and the specifications of the model elements can be used as test oracles.
- The model can be used to guide selective regression testing of the system as it evolves in maintenance [13].

The software testing literature offers a variety of techniques that can be applied or adapted in a reasonably straightforward way to these kinds of testing (e.g., by defining architecture-oriented structural coverage criteria, defining the architectural equivalent of top-down or bottom-up integration testing, etc.). Yet there is very little in the testing literature that addresses the unique testing challenges posed by component-based software.

Distributed component-based systems of course exhibit all of the well-known problems that make testing "traditional" distributed and concurrent software difficult. But testing of component-based software (distributed or otherwise) is further complicated by *technological heterogeneity* and *enterprise heterogeneity* of the components used to build systems. Technological heterogeneity refers to the fact that different components can be programmed in different programming languages and for different operating system and hardware platforms, meaning that testing a component-based system may require a testing method that operates on a large number of languages and platforms. Enterprise heterogeneity refers to the fact that off-the-shelf components can be provided by different, possibly competing suppliers, meaning that no one supplier has complete control over or complete access to the development artifacts associated with each component for purposes of testing. And in the most extreme situations of *dynamic evolution*, components can be replaced within, added to, and deleted from a running system, potentially forgoing a traditional period of testing prior to deployment of the new configuration [5,8].

Thus, in this author's view, it is these problematic characteristics of component-based software that raise the

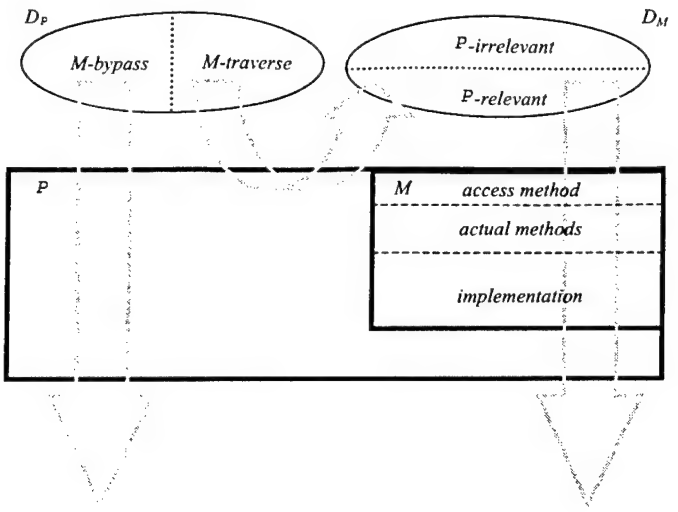


Fig. 1 . A Model of Component-Based Software.

most interesting and important challenges in exploiting architectural models for software testing.

2 A FORMAL DEFINITION OF COMPONENT-BASED TEST ADEQUACY

Any attempt to develop a foundation for testing component-based software must begin by establishing an appropriate formal model of test adequacy. A *test adequacy criterion* is a systematic criterion that is used to determine whether a test suite provides an adequate amount of testing for a program under test. Previous definitions of adequacy criteria have defined adequate testing of a program independently of any larger system that uses the program as a component. This perhaps may be due to the traditional view of software as a monolithic code base that can be put through several phases of testing prior to its deployment, and by the same organization that built the software in the first place. However, a test set that satisfies a criterion in the traditional sense might not satisfy the criterion if it were interpreted with respect to the subset of the component's functionality that is used by a larger system.

Consider the simple example of the *statement coverage* criterion, which requires a test set to exercise each statement in the component under test at least once. There may be a large number of elements in the component's input domain that could be chosen to cover a particular statement. However, the element that is ultimately chosen may not be a member of that subset of the input domain that is utilized by the larger program using the component. Hence, while according to traditional notions of test adequacy the test case could serve as a member of an adequate test set for the component, from the perspective of the larger program using the component, the test set would be inadequate.

In a recent paper, the author developed a formal model of component-based software and a formal definition of component-based test adequacy [12]. This work attempts to capture in a formal way the need to consider the context in which a component will be used in order to judge whether or not the component, and the system using the component, has been adequately tested.

Fig. 1 represents pictorially the formal model of component-based software. The figure illustrates a program P containing a constituent component M. In general, P may contain several such components M, and P may itself be a component within some larger system. As shown in the figure, M is viewed as declaring in its interface a single *access method* that handles the invocation of the *actual methods* of M. For each parameter of an actual method of M, there is a corresponding parameter of the same type and mode in the access method. The access method includes an additional parameter used to identify the actual method that is to be invoked. The input domain of M is then the input domain of its access method, which is the union of the input domains of the actual methods of M, but with each element extended with the appropriate method identifier.

Let D_P be the input domain of P, and let D_M be the input domain of the access method of M. As shown in the figure, there are four important subsets of these input domains, which are defined formally as follows:

Definitions:

$$M\text{-traverse}(D_P) = \{\, d \in D_P \mid \text{execution of } P \text{ on input } d \text{ traverses } M \,\}$$

$$M\text{-bypass}(D_P) = D_P - M\text{-traverse}(D_P)$$

$P\text{-relevant}(D_M)$ =
$\{ d \in D_M \mid \exists d' \in M\text{-traverse}(D_P) \bullet$ execution of P on input d' traverses M with input $d \}$

$P\text{-irrelevant}(D_M) = D_M - P\text{-relevant}(D_M)$

The phrase "execution of P traverses M" is taken to mean that the execution of P includes at least one invocation of M's access method. The *M-traverse* subset of D_P is then the set of all inputs of P that cause the execution of P to traverse M. The $P\text{-relevant}$ subset of D_M is the set of all inputs of M's access method that P uses for its traversals of M. The *M-bypass* subset of D_P is the set of all inputs of P that cause the execution of P to "bypass" or avoid traversing M. Finally, the $P\text{-irrelevant}$ subset of D_M is the set of all inputs of M's access method that P never uses for its traversals of M.[1]

The formal definition of test adequacy for component-based software is developed in terms of applicable subdomain-based test adequacy criteria, as defined by Frankl and Weyuker [2]. In particular, a test adequacy criterion C is *subdomain-based* if there is a nonempty multiset $\mathcal{SD}_C(D)$ of subdomains of D (the input domain of the program under test), such that C requires the selection of one test case from each subdomain in $\mathcal{SD}_C(D)$.[2] Furthermore, C is *applicable* if the empty subdomain is not an element of $\mathcal{SD}_C(D)$ [2]. Thus, a test set is *C-adequate* if and only if it contains at least one test case from each subdomain in $\mathcal{SD}_C(D)$. Since testers rarely satisfy 100% of the test requirements induced by a test adequacy criterion, it also makes sense to say that a test set is *n% C-adequate* if it contains at least one test case from n percent of the subdomains in $\mathcal{SD}_C(D)$. These definitions capture the traditional notion of test adequacy, and they make no distinction between a program and a component.

In order to define test adequacy for component-based software, it is necessary to first partition the subdomains induced by an applicable subdomain-based criterion C according to the partitioning of D_P and D_M:

Definitions:

$\mathcal{SD}_C(M\text{-traverse}(D_P)) =$
$\{ D \subseteq M\text{-traverse}(D_P) \mid \exists D' \in \mathcal{SD}_C(D_P) \bullet D \subseteq D'$ and $D'-D \subseteq M\text{-bypass}(D_P)$ and $D \neq \emptyset \}$

$\mathcal{SD}_C(M\text{-bypass}(D_P)) =$
$\{ D \subseteq M\text{-bypass}(D_P) \mid \exists D' \in \mathcal{SD}_C(D_P) \bullet D \subseteq D'$ and $D'-D \subseteq M\text{-traverse}(D_P)$ and $D \neq \emptyset \}$

$\mathcal{SD}_C(P\text{-relevant}(D_M)) =$
$\{ D \subseteq P\text{-relevant}(D_M) \mid \exists D' \in \mathcal{SD}_C(D_M) \bullet D \subseteq D'$ and $D'-D \subseteq P\text{-irrelevant}(D_M)$ and $D \neq \emptyset \}$

$\mathcal{SD}_C(P\text{-irrelevant}(D_M)) =$
$\{ D \subseteq P\text{-irrelevant}(D_M) \mid \exists D' \in \mathcal{SD}_C(D_M) \bullet D \subseteq D'$ and $D'-D \subseteq P\text{-relevant}(D_M)$ and $D \neq \emptyset \}$

Note that according to these definitions, each subdomain induced by criterion C on program P is partitioned into its *M-traverse* subset and its *M-bypass* subset. Note that the definitions discard empty subdomains, in order to retain the applicability of C.

Given the above definitions, adequate testing of component-based software can now be formally defined. First, the concept *C-adequate-for-P* is defined to characterize adequate *unit testing* of M:

Definition (C-adequate-for-P): A test set T_M is *C-adequate-for-P* if it contains at least one test case from each subdomain in $\mathcal{SD}_C(P\text{-relevant}(D_M))$.

Second, the concept *C-adequate-on-M* is defined to characterize adequate *integration testing* of P with respect to its usage of M:

Definition (C-adequate-on-M): A test set T_P is *C-adequate-on-M* if it traverses M with at least one element from each subdomain in $\mathcal{SD}_C(P\text{-relevant}(D_M))$.

These definitions can be extended as before to accommodate a notion of percentage of adequacy.

Note that although it is P that is being tested in integration testing, these definitions require the criterion C to be chosen and then evaluated *in terms of M* in order to ensure adequate testing of the relationship between P and M. For example, C could be a criterion that requires each of the actual methods of M to be exercised at least once. This is a reasonable requirement for adequate integration testing of P, and the definition of *C-adequate-on-M* ensures that the criterion would be interpreted only with respect to the methods of M that P invokes anywhere in its source code. Of course, C need not be the same criterion as the one used to design T_P in the first place; it merely imposes a requirement on the testing achieved by T_P.

There are a number of additional interesting consequences of these definitions. For instance, a test set T_M that is *C-adequate* might not be *C-adequate-for-P*, and vice versa. Furthermore, a test set T_P that is *C-adequate* might not be *C-adequate-on-M*, and vice versa. And similar statements can be made with respect to percentage of adequacy.

[1] Note that this model fails to account for the possibility of non-determinism in the execution of P or M.

[2] An example of such a criterion is *statement coverage*, which induces one subdomain for each executable statement in a program, with each subdomain containing exactly those inputs that cover its associated statement.

3 ADEQUATE TESTING AND SOFTWARE ARCHITECTURES

The formal model presented above provides an initial foundation for studying and evaluating test adequacy for component-based systems. However, there are two important issues that merit consideration. One issue is the practical applicability of the model. It is one thing to argue that components must be tested with respect to the context in which they will be used. It is another thing to determine how this will be accomplished, especially in the presence of off-the-shelf components, whose important attributes needed for evaluation of test adequacy criteria (such as input domain, specification, implementation structure, etc.) may be difficult or impossible to ascertain.

A second issue is to determine how the model relates to, and can be adapted to, software architectural modeling. One approach is to view formal architectural models as inducing definitions of input domains for architectural elements, and then applying the model to these induced input domains. However, this approach must take into consideration at least three important attributes of architectural models and the ADLs used to specify them.

First, many ADLs support specification of other kinds of architectural elements in addition to components, such as connectors and configurations (see Medvidovic et al. for a complete discussion of ADL features and capabilities [6,7]). The formal model presented above must be enhanced to take these additional kinds of elements into account. In some ADLs these elements can possibly be treated as components for the purpose of testing. For instance, the connectors in Wright encapsulate behavior and provide a static, finite collection of interface elements [1]. However, in other ADLs such elements differ substantially from components. For instance, a connector in C2 does not have an interface per se, but instead is *contextually reflective* of the interfaces of the components that it connects [15]. Furthermore, this set of components can vary dynamically [8].

Second, many ADLs distinguish between the *conceptual architecture* that is formally modeled in an ADL and the *implementation architecture* of the actual system, and different ADLs impose different requirements on the relationship between the two. At a minimum, this distinction means that the input domains of the conceptual architecture may differ from those of the implementation architecture. In particular, the input domain of a component in the conceptual architecture can be a (proper) subset of the input domain of the implementation-level component or components that implement the conceptual component. This is to be expected especially in situations where an off-the-shelf component provides more functionality than is needed by a system that uses it. Thus, the formal model of test adequacy developed above must be enhanced to account for the additional complexities introduced by domain relationships between conceptual and implementation architectures.

Third, ADLs support explicit representation of a rich variety of behavioral relationships and other dependencies between architectural elements [14]. These relationships do not always strictly conform to a caller/callee style of component relationships as depicted in Fig. 1, and it may not be possible to characterize them fully and precisely in terms of the input domains of related elements. Thus, the formal model of test adequacy developed above must be enhanced to account for the richness of inter-element relationships.

With these enhancements in place, the formal model of test adequacy can be used in conjunction with formal architectural models to guide testing of software in a manner that is truly adequate. A key challenge in incorporating such enhancements is to address the broad range of semantic models and modeling and analysis concerns of the many ADLs that have been defined.

4 CONCLUSION

This paper has discussed challenges in exploiting architectural models for software testing, with the discussion framed in terms of the author's recent work on defining a formal model of test adequacy for component-based software. An explicit architectural viewpoint in software engineering offers the promise of dramatically improving—and in the process altering— the way software is developed [10]. While these changes will not obviate the need for testing, one can at least attempt to find ways of exploiting formal architectural models for the purpose of testing. While architectural models offer a rich source of information to support testing, any attempt to exploit architectural models for testing must be cognizant of the unique characteristics of the new kinds of systems that an architectural viewpoint engenders.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973, and by the Air Force Office of Scientific Research under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

1. R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

2. P.G. Frankl and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods", *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202–213, 1993.

3. D. Hamlet, "Software Component Dependability—a Subdomain-based Theory", RST Corporation, Technical Report RSTR-96-999-01, September 1996.

4. D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, 1995.

5. J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

6. N. Medvidovic and D.S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages", *Proc. USENIX Conference on Domain Specific Languages*, Santa Barbara, CA, pp. 199–212, 1997.

7. N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, 1997.

8. P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution", *Proc. 20th International Conference on Software Engineering*, Kyoto, Japan, 1998.

9. P. Oreizy, N. Medvidovic, R.N. Taylor, and D.S. Rosenblum, "Software Architecture and Component Technologies: Bridging the Gap", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA January 1998.

10. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

11. D.J. Richardson and A.L. Wolf, "Software Testing at the Architectural Level", *Proc. Second International Software Architecture Workshop*, San Francisco, CA, pp. 68–71, 1996.

12. D.S. Rosenblum, "Adequate Testing of Component-Based Software", Department of Information and Computer Science, University of California, Irvine, Irvine, CA, Technical Report 97-34, August 1997.

13. G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

14. J.A. Stafford, D.J. Richardson, and A.L. Wolf, "Chaining: A Software Architecture Dependence Analysis Technique", Department of Computer Science, University of Colorado, Boulder, CO, Technical Report CU-CS-845-97, September 1997.

15. R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

from the Proceedings of the Third International Software Architecture Workshop (ISAW-3), Lake Buena Vista, FL, November 1–2, 1998

Merging Component Models and Architectural Styles

Rema Natarajan David S. Rosenblum

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA

{rema,dsr}@ics.uci.edu

1. ABSTRACT

Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of component interoperability models, such as ActiveX, CORBA and JavaBeans. In academia, there has been intensive research in developing a notion of software architecture. Our research involves studying how standard component models can be extended to accommodate important issues of architecture, including a notion of architectural style and support for explicit connectors. In this paper, we discuss issues arising from our initial effort in this research, where we have extended the JavaBeans component model to support component composition according to the C2 architectural style.

1.1 Keywords

Architectural style, C2, component standards, connectors, JavaBeans, software architecture

2. INTRODUCTION

Components have increasingly become the unit of development of software. In industry, there has been considerable work in the development of component interoperability models, such as ActiveX [1], CORBA [4], and JavaBeans [6]. These models help developers deal with the complexity of software and promote reuse. Component interoperability models also make a positive move toward standardization of components, and the creation of a software component marketplace.

Software architecture research deals with the same issues of software complexity and promoting reuse. Software architecture has been the focus of intense research in academia. Architectures help developers focus on system level requirements and the interconnection of components in a large-scale software system.

Both these approaches use software components as the building blocks. With component interoperability models, the focus is on specifying interfaces, packaging, binding mechanisms, inter-component communication protocols, and expectations regarding the runtime environment. With software architectures and architectural styles, the focus in on specifying systems of communicating components, analyzing system properties, and generating "glue" code that binds system components [3].

It seems intuitive to merge component interoperability models with suitable architectural styles to leverage the full benefit from both technologies, and to develop a comprehensive approach to software development. It also opens up opportunities for researchers in industry and in academia to exchange views and results.

In this paper, we describe our work in enhancing the JavaBeans component model to support component composition according to the C2 architectural style. Our approach enables the design and development of applications in the C2 architectural style using off-the-shelf Java components or *beans* that are available to the developer. The creation of individual components with their specific interfaces, functionalities and behaviors is a different task from the composition of an architecture of a system that satisfies requirements. The merging of the component interoperability model with the architectural style provides a seamless integration of both activities.

3. THE JAVABEANS COMPONENT MODEL

The JavaBeans component model is a component model tailored to the Java language. The JavaBeans design pattern defines a protocol to which beans must adhere. This interface pattern mainly consists of the properties, methods, and events that together define a bean interface. *Properties*

ISAW3 Orlando Florida USA

encapsulate key attributes of a bean and can be read-only, read/write, *bound* (meaning they generate events whenever they change values) or *constrained* (meaning their changes can be vetoed by other beans). *Methods* are public operations that form part of the bean interface. Beans communicate with each other through bean *events*; the event handling is based on the Java 1.1 event model. Thus, the JavaBeans component model concentrates on the interface a Java software building block can or should represent. It does not specify how the building blocks can or should be combined to create any type of application. It specifies how two or more beans can communicate information, without imposing any semantic rules on the information exchanged or on the topology of any bean communication network [6]. The JavaBeans design pattern is designed to make beans tool-aware; in particular, the interface pattern has been defined for a modern software developer who will manipulate beans via visual interactions.

4. THE C2 ARCHITECTURAL STYLE

The C2 architectural style is primarily concerned with high-level system composition issues, rather than particular component packaging approaches [3,5]. The building blocks of C2 architectures are components (computational elements) and connectors (interconnection and communication elements). This separation of computation from communication enables the construction of flexible, extensible, and scalable systems that can evolve both before and during runtime. This style places no restrictions on the implementation language or granularity of components and connectors, potentially allowing it to use multiple interoperability technologies for its connectors. This flexibility has enabled us to use the event-based interoperability of JavaBeans for our purposes. Central to the C2 style is the principle of limited visibility or "substrate independence": components are arranged in a layered fashion in a C2 architecture, and a component is completely unaware of components that reside beneath it in the stack of component layers. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. Components communicate only by exchanging messages through connectors, which greatly simplifies the problem of control integration issues; this property also facilitates low-cost interchangeability of components to construct different members of the same system family. Two components cannot assume that they will execute in the same address space; this eliminates complex dependencies, such as components sharing global variables and simplifies modification of architectures. Conceptually, components run in their own thread(s) of control, allowing components with different threading models to be integrated into a single application. Finally, a conceptual C2 architecture can be instantiated in a number of different ways. Many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation techniques.

5. EXPERIENCE TO DATE

We have begun our investigation of the problem of merging component models with architectural styles by enhancing the BeanBox (the visual composition environment for JavaBeans) that comes with Sun's Bean Development Kit. The BeanBox allows developers to develop beans using the beans design pattern, and instantiate and test the beans in the BeanBox. Our enhancements make the BeanBox C2-aware. In particular, the enhanced BeanBox allows one to build complex compositions of the beans in the C2 style as different instantiations of a given C2 architecture. It is not necessary to do any translation or mapping to convert an existing bean into a C2 component. Introspection mechanisms employed in the BeanBox are used to extract the properties, methods and events that form the public interface of the bean. Conceptually, beans communicate using bean events; these events then become the requests and notifications in the C2 architecture. The developer informs the Beanbox through an appropriate dialog about the events that are to be classified as requests and events. An instantiated bean is wrapped in a C2 Component, which has a wrapper for the bean, and a dialog manager that manages the communication of beans through these requests and notifications.

As shown in Fig. 1, this wrapping is done according to the general model of wrapping that has been defined for components in the C2 style [5]. The visual interface of the BeanBox allows the developer to build C2 architectures composed of beans intuitively and easily. The tool automatically enforces the C2 style constraints and notifies the developer when C2 style constraints are violated. C2 connector beans are used as the connectors in the architecture.

A key advantage of this approach is that our architectural infrastructure is now complete, in the sense that the full range of developmental activities is supported from the design, development and testing of individual components, to the design, development and testing of architectures that are compositions of these individual elements. Another advantage is that all these activities are now integrated into a single environment, and this leads the way to a seamless, comprehensive development philosophy that facilitates easy shifting of focus from one activity to another. Sophisticated architectural development tools built along these lines will tie in neatly with component-based software development.

6. CONCLUSIONS

Having considered and explored the possibility of combining a popular component interoperability model with a useful software architectural style, we are convinced of the advantages of this approach to development of

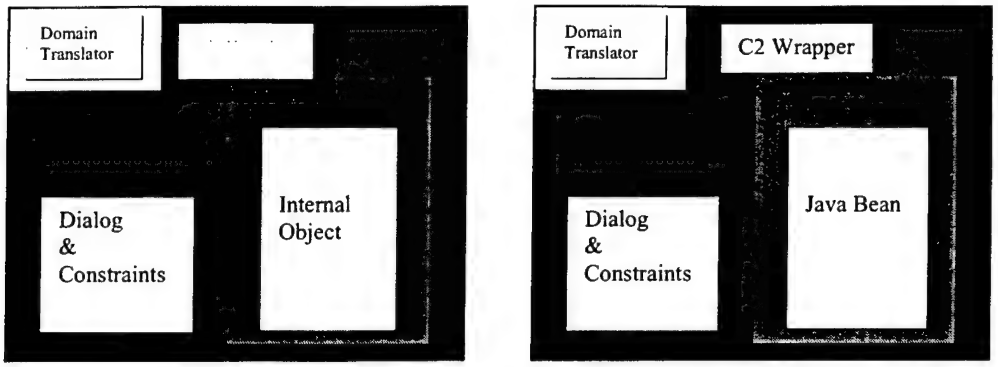


Fig. 1. Wrapping of C2 components; the general model of wrapping is shown in the picture on the left, while the picture on the right shows how the general model has been applied for JavaBeans.

component-based software. In the future, we plan to complete the implementation of this tool, and we plan to further investigate the issues raised and opportunities opened up by this approach. For example, this research opens up interesting possibilities to use an enhanced BeanBox to test runtime behaviors of system architectures before system implementation is completed.

The plug-in capabilities of the JavaBeans environment, and the philosophy of substrate independence in C2, make substituting of components and rearranging of architectures fairly easy. The BeanBox is an example of a tool where the distinction between the design environment and the runtime environment of systems has become blurred. This is an issue that is being studied in greater depth in other work on C2, in the context of designing and instantiating system architectures [2]. Our experience, we believe, will help us expand and develop our understanding of the synergy between component models and software architectures.

7. ACKNOWLEDGMENTS

Discussions with Dick Taylor and Peyman Oreizy helped us improve many of the ideas presented in this paper. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973, and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

8. REFERENCES

[1] D. Chappell, *Understanding ActiveX and OLE*. Redmond, WA: Microsoft Press, 1996.

[2] N. Medvidovic, P. Oreizy, and R.N. Taylor, "Reuse of Off-the-Shelf Components in C2-Style Architectures", *Proc. 19th International Conference on Software Engineering*, Boston, MA, pp. 692–700, 1997.

[3] P. Oreizy, N. Medvidovic, R.N. Taylor, and D.S. Rosenblum, "Software Architecture and Component Technologies: Bridging the Gap", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA January 1998.

[4] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[5] R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[6] L. Vanhelsuwe, *Mastering JavaBeans*: SYBEX Inc, 1997.

from the Proceedings of the ICSE '98 International Workshop on the Principles of Software Evolution (IWPSE '98), Kyoto, Japan, April 20–21, 1999

An Architecture-Based Approach to Software Evolution

Nenad Medvidovic
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-078, USA
+1-213-740-5579
neno@usc.edu

David S. Rosenblum
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, USA
+1-949-824-6534
dsr@ics.uci.edu

Richard N. Taylor
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, USA
+1-949-824-6429
taylor@ics.uci.edu

1. INTRODUCTION

In order for large, complex, multi-lingual, multi-platform, long-running systems to be economically viable, they need to be evolvable. Support for software evolution includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires support for reuse of third-party components. The costs of system maintenance (i.e., evolution) are as high as 60% of the overall development costs [6]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. Traditional development approaches (e.g., structural programming or object-oriented analysis and design) have in particular failed to properly decouple computation from communication within a system, thus supporting only limited reconfigurability and reuse. Evolution techniques have also typically been programming language (PL) specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, multi-lingual, multi-platform components that originate from multiple sources.

In this paper, we posit that an explicit architectural focus can remedy many of these difficulties and enable flexible construction and evolution of large systems. Software architectures present a high level view of a system, enabling developers to abstract away the irrelevant details and focus on the "big picture." Another key is their explicit treatment of software connectors, which separate communication issues from computation in a system. However, existing architecture research has thus far largely failed to take advantage of this potential for adaptability, for two reasons:

- connectors are often not treated explicitly or, when they are, they are too rigid and do not accommodate modification of their attached components easily; and
- no specific techniques have been developed to support flexible architecture-based design and evolution.

Three distinct building blocks of a software architecture are components, connectors, and architectural configurations (topologies) [13]. Each of them may evolve. Our work to date has focused on the evolution of individual components and architectural configurations. In the future, we intend to investigate the proper techniques for evolving connectors. For evolving individual components, our approach expands the traditional techniques for supporting evolution (e.g., modularity, typing). We introduce explicit, flexible connectors to aid the evolution of architectural configurations.

The following section discusses our approach to component evolution and introduces an architectural type theory on which the approach is based. Section 3 discusses the role of software connectors in the evolution of architectural configurations. Conclusions and a discussion of ongoing work round out the paper.

2. COMPONENT EVOLUTION

Researchers in software architectures, and particularly in architecture description languages (ADLs), can learn from extensive experience in the area of PLs. For example, an existing software module can evolve in a controlled manner via subtyping. Our approach to component evolution is indeed based on type theory. We treat each component in an architecture as a type and support its evolution via subtyping. However, while PLs (and several existing ADLs [4, 5, 7]) support a single subtyping method, architectures may require multiple subtyping methods, many of which are not commonly supported in PLs. Therefore, an extension to PL type theory is needed.

A useful overview of PL subtyping mechanisms is given by Palsberg and Schwartzbach [15]. They describe a consensus in the object-oriented (OO) typing community regarding the definition of a range of OO typing mechanisms. *Arbitrary subclassing* allows any class to be declared a subtype of another, regardless of whether they share a common set of methods. *Name compatibility* demands that there exist a shared set of method names available in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass must preserve the interface of the superclass. *Behavioral conformance* allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the

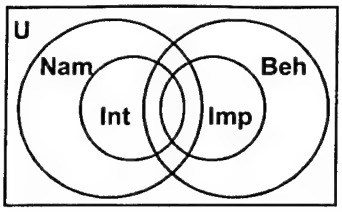


Figure 1. A framework for understanding OO subtyping mechanisms as regions in a space of type systems.

supertype. *Strictly monotone subclassing* also demands that the subtype preserve the particular implementations used by the supertype.

We have developed a framework for understanding these subtyping mechanisms as regions in a space of type systems, shown in Fig. 1. The entire space of type systems is labeled *U*. The regions labeled *Int* and *Beh* contain systems that demand that two conforming types share interface and behavior, respectively. The *Imp* region contains systems that demand that a type share particular implementations of all supertype methods, which also implies that types preserve the behavior of their supertypes. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance. Each subtyping mechanism described in [15] and summarized above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the Int and Beh regions and is expressed as *int* **and** *beh*.

We have demonstrated the utility of such a flexible subtyping mechanism in our previous work, where we have encountered numerous situations in which new components were created by preserving one or more aspects of one or more existing components [10, 12]. Several examples are shown in Fig. 2.

2.1 Architectural Type Theory

In [10] we discussed the types of syntactic constructs needed in an ADL in order to support this approach. In this section we present a brief overview of the underlying type theory. The two possible applications of an architectural type theory are type checking of architectural descriptions and evolution of existing components by software architects. Each is briefly discussed below.[1]

Every component is an *architectural type*. An architectural type, AT, has a name, a set of interface elements, an associated behavior, and (possibly) an implementation:

AT = <nam, int*, beh, imp>

Each interface element has a direction indicator (provided or required), a name, and a set of parameters. Each parameter, in turn, has a name and a type.

[1] We omit some details for simplicity.

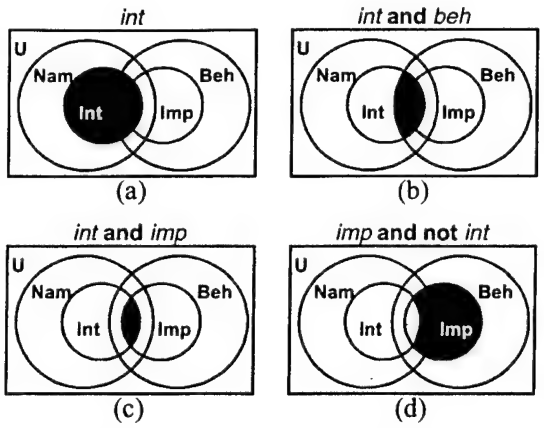


Figure 2. Examples of component subtyping mechanisms we have encountered in software architectures: *(a)* interface conformance; *(b)* behavioral conformance; *(c)* strictly monotone subclassing; *(d)* implementation conformance with a different interface (e.g., software adaptors [18]).

int = <dir, int_nam, param*>
param = <param_nam, param_type>

A component's behavior consists of an invariant and a set of operations. The invariant is used to specify any protocol constraints on the use of the component. Each operation has (a set of) preconditions and postconditions and (possibly) a result.

beh = <inv, oper*>
oper = <pre, post, result>

Finally, since we separate the interface from the behavior, we define a function, f, that maps every interface element to an operation of the behavior. This function is "onto": each operation exports at least one interface.

A subtyping relationship, $\leq$, between two components, C_i and C_j, is defined as a disjunction of *nam*, *int*, *beh*, and *imp* relationships (see Fig. 1):

$$(\forall C_i, C_j{:}AT)(C_j \leq C_i \Leftrightarrow C_j \leq_{nam} C_i \vee C_j \leq_{int} C_i \vee C_j \leq_{beh} C_i \vee C_j \leq_{imp} C_i)$$

We consider *int* and *beh* subtyping relationships in more detail below. *Nam* is a trivial relationship; we have encountered it in practice only as part of the stronger *int* relationship. Similarly, although useful in practice for evolving components, *imp* is not a particularly interesting relationship from a type-theoretic point of view. Implementation conformance can be established with a simple syntactic check: the operations of the subtype must have the same implementation as the corresponding operations of the supertype.

Component C_j is an *interface subtype* of C_i if and only if it provides at least (but not necessarily only) the interface elements provided by C_i with identical names and direction

indicators, and *matching* parameters for each interface element. Two parameters belonging to the two components' interface elements match if and only if they have identical names and the parameter type of C_i is a subtype of the parameter type of C_j (*contravariance of arguments*). Note that, as with interface elements, the subtype must provide at least (but not necessarily only) the parameters that match the supertype's parameters:[2]

$$(\forall C_i, C_j{:}AT)(C_j \leq_{int} C_i \Leftrightarrow (\forall M \in C_i.int)(\exists N \in C_j.int)$$
$$((M.dir = N.dir) \wedge (M.int\_nam = N.int\_nam) \wedge$$
$$((\forall P_m \in M.param)(\exists P_n \in N.param)$$
$$((P_m.param\_nam = P_n.param\_nam) \wedge$$
$$(P_m.param\_type \leq P_n.param\_type))))$$

A *behavioral subtyping* relationship between two components is specified as follows:

$$(\forall C_i, C_j{:}AT)(C_j \leq_{beh} C_i \Leftrightarrow (C_j.beh.inv \Rightarrow C_i.beh.inv) \wedge$$
$$(\forall P \in C_i.beh.oper)(\exists Q \in C_j.beh.oper)$$
$$((P.pre \Rightarrow Q.pre) \wedge (Q.post \Rightarrow P.post) \wedge$$
$$(Q.result \leq P.result))$$

This definition requires that invariant of the supertype be ensured by that of the subtype. Furthermore, each operation of the supertype has a corresponding operation in the subtype, where the subtype's operation has the same or weaker preconditions, same or stronger postconditions, and the type of its result is a subtype of the supertype's result type (*covariance of result*).

The subtyping relationship expressed by the combination of these two definitions and the mapping function, f, results in the region depicted in Fig. 2b and is similar to other researchers' notions of behavioral subtyping (e.g., America [2], Liskov and Wing [8], Leavens et al.[3]). However, in these approaches type correctness is characterized as either legal or illegal. In software architectures, various degrees of type conformance may be acceptable, so that, for example, the interfaces of two communicating components may match up only partially. Additionally, by separating interface from behavior (and adding the nam and imp relationships), we give a software architect more latitude in choosing the direction in which to evolve a component. Such a flexible type system allows some potentially undesirable side effects (e.g., a supertype and its subtype may not always be interchangeable in a given architecture). However, it is left up to the architect to decide whether he wants to preserve architectural type correctness or simply enlarge his palette of design elements, which could then be used in the future.

3. CONFIGURATION EVOLUTION

We employ flexible connectors to support the evolution of architectural configurations. Connectors remove from components the responsibility of knowing how they are interconnected. Connectors also introduce a layer of indirection between components. The potential penalties paid due to this indirection (e.g., performance) should be outweighed by other benefits of connectors, such as their role as facilitators of evolution. To facilitate architectural evolution, connectors must be flexible, i.e., they must easily accommodate changes to their attached components. At a minimum, these changes include component addition, removal, replacement, and reconnection.

Existing approaches tend to sacrifice the potential flexibility introduced by connectors in order to support more powerful architectural analyses. For example, Wright [1] and UniCon [16] require the architect to specify the types of component *ports* and *players*, respectively, that can be attached to a given connector *role*. Furthermore, although some variability is allowed in specifying the number of components that a given connector will be able to support (parameterized number of roles in Wright; potentially unbounded number of players with which each role may be associated in UniCon), once these variables are set at architecture specification time, neither approach allows their modification.

Our approach to configuration evolution is based on our experience with the C2 architectural style [17]. In the C2 style, connectors are communication message routing devices. To provide an added degree of freedom in composing components, C2 connectors support implicit invocation, which minimizes component interdependencies. We have demonstrated that C2 connectors provide strong support for evolution of architectural configurations both at specification time [11, 12] and at runtime [9, 14].

A unique aspect of C2 connectors, and a direct facilitator of architectural evolution, are their *context-reflective interfaces*. A connector does not export a specific interface. Instead, it acts as a communication conduit which, in principle, supports communication among any set of components. The number of connector ports is not predetermined, but changes as components are attached or detached. The "interface" exported by a C2 connector is thus a function of the attached components' interfaces. This allows any C2 connector to support arbitrary addition, removal, replacement, and reconnection of components or other connectors.

Clearly, it is not always the case that two components can communicate (e.g., due to mismatched interfaces, or message filtering), even though they may be attached to the same connector. At the architectural level, this can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into implementation, implicit invocation guarantees that, in the worst case, communication messages will be lost (*partial-* or *no-communication* [12, 17]), but the rest of the system's architecture will be able to perform at least in a degraded mode.

---

[2] In our notation, "X.Y" denotes X's constituent Y. For example, "C_i.int" denotes component C_i's interface.

4. CONCLUSIONS AND FUTURE WORK

Software architectures show great potential for reducing development costs while improving the quality of the resulting software. Architectures also provide a promising basis for supporting software evolution. However, improved evolvability cannot be achieved simply by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures, and ADLs in particular, as tools which also must be supported with specific techniques to achieve desired properties. In this paper, we have outlined two such techniques for supporting evolution, one for components and the other for architectural configurations.

We have already put a subset of these ideas into practice in the context of the C2 style and its accompanying ADL. We are currently developing a set of tools to support architectural subtyping, type checking, and mapping of architectural descriptions to the C2 implementation infrastructure [11]. We are also expanding C2 connectors to support more complex message passing protocols, as well as existing middleware technologies (e.g., CORBA and Java's RMI). A number of issues remain items of future work. These include investigation of techniques for evolving connectors, application of the type theory to other ADLs and across multiple levels of architectural refinement, further research of issues in adapting and adopting legacy components into architectures using the subtyping approach, automating the evolution of existing components to populate partial architectures, and assessment of the applicability of the properties of C2 connectors described in Section 3 to other architectural approaches.

5. ACKNOWLEDGEMENTS

Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-97-2-0021 and F30602-94-C-0218, and by the Air Force Office of Scientific Research under grant number F49620-98-1-0061. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973.

6. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, volume 489, pages 60-90, Springer-Verlag, 1991.

[3] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

[5] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

[6] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[7] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.

[8] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[9] N. Medvidovic. ADLs and Dynamic Architecture Changes. In Alexander L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pages 24-27, San Francisco, CA, October 14-15, 1996.

[10] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 16-18, 1996.

[11] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 17-23, 1997.

[12] N. Medvidovic and R.N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, pages 237-248, October-December 1997.

[13] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 60-76, Zurich, Switzerland, September 22-25, 1997.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. To appear in *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 19-25, 1998, Kyoto, Japan. Also available as Technical Report, UCI-ICS-97-39.

[15] J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992.

[16] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pages 314-335, April 1995.

[17] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.

[18] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, pages 176-190, Portland, OR, USA, October 1994.

from the Proceedings of the Third International Software Architecture Workshop (ISAW-3), Lake Buena Vista, FL, November 1–2, 1998

Issues in Supporting Event-based Architectural Styles

Antonio Carzaniga
Univ. of Colorado at Boulder
Dept. of Computer Science
Campus Box 430
Boulder, CO 80309-0430
carzanig@cs.colorado.edu

Elisabetta Di Nitto
CEFRIEL and UCI
CEFRIEL
Via Fucini, 2
20133 Milano, Italy
dinitto@elet.polimi.it

David S. Rosenblum
Univ. of California, Irvine
Dept. of Inf. and Comp. Science ICS2 209
Irvine, CA 92697-3425
dsr@ics.uci.edu

Alexander L. Wolf
Univ. of Colorado at Boulder
Dept. of Computer Science
Campus Box 430
Boulder, CO 80309-0430
alw@cs.colorado.edu

1. INTRODUCTION

The development of complex software systems is demanding well established approaches that guarantee robustness of products, economy of the development process, and rapid time to market. This need is becoming more and more relevant as the requirements of customers and the potential of computer telecommunication networks grow.

To address this issue, researchers in the field of software architecture are defining a number of languages and tools that support the definition and validation of the architecture of systems. Also, a number of *architectural styles* are being formalized. Each of them defines "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or a subsystem, together with local or global constraints on the way the composition is done" [5]. The formalization of styles helps the understanding and categorization of existing architectures and supports developers in the definition of the structure of new systems.

A style that is very prevalent for large-scale distributed applications is the *event-based style*. In an event-based style, components communicate by generating and receiving *event notifications*. A component usually generates an event notification when it wants to let the "external world" know that some relevant event occurred either in its internal state or in the state of other components with which it interacts. When an event notification is generated, it is propagated to any component that has declared interest in receiving it. The generation of the event notification and its propagation are performed asynchronously. Usually, a connector called an *event service* (or an *event dispatcher* or *bus*) is in charge of managing the propagation of the event notifications. This propagation is completely hidden to the component that generated the event. Thus, the event service implements a multicasting mechanism that fully decouples event generators from event receivers. This provides two important effects:

- A component can operate in the system without being aware of the existence of other components. All it has to know is the structure of the event notifications that are interesting to it.
- It is always possible to plug a component in and out of the architecture without affecting the other components directly.

These two effects guarantee a high compositionality and reconfigurability of a software architecture.

In the last few years, interest in the event-based style among practitioners has resulted in the development of a number of *event-based middleware infrastructures* (see for instance [3], [7], and [6]). These infrastructures implicitly support the event-based style; that is, they provide APIs and frameworks for defining applications structured according to this style.

We started investigating the event-based style two years ago in two separate research efforts where we participated in the definition of a general model for event-based architectures [4] and in the development an event-based infrastructure called JEDI [1]. By using JEDI and by comparing it with other systems and infrastructures, we recognized a number of different variations of the event-based style. These variations have different impact on the structure, the behavior, and the performance (in other words, on the architecture) of applications. Therefore, these variations need to be carefully analyzed and explicitly defined as part or specialization of the event-based style, in order to be exploited whenever the architecture of a system is defined.

In this paper we identify the event-based style variations introduced by a number of event-based middleware infrastructures and point out the advantages and drawbacks of the different approaches as well as the open issues.

2. EVENT-BASED STYLE AND MIDDLEWARE INFRASTRUCTURES

Figure 1 shows an operational and pragmatic description of the event-based style. An architecture that realizes this style is characterized by a connector called an *event service*. It is in charge of dispatching event notifications. The

ISAW3 Orlando Florida USA

components can be classified into two categories: *recipients* and *objects of interest*. Recipients declare their interest in receiving event notifications by issuing a *subscribe* operation offered by the event service. Objects of interests notify the occurrence of an event by sending a *publish* request to the event service. Alternatively, the event service itself can poll objects of interests to know if some event has been produced. A component can behave both as an object of interest and a recipient of events. The event service reacts to a publish request by forwarding the corresponding event notification to all the recipients that have subscribed to it. This high-level architectural style is being exploited by most of the event-based infrastructures that have been currently implemented. In the following we mention a few of these that provide significant variations of the style.

CORBA defines the concept of *channel*, which is a simplified version of an event service [3]. All the recipients that are connected to a channel receive all the notifications that are published by object of interests on that channel.

Smartsockets [7] proposes a more powerful approach in which the event service can accept subscriptions for a number of different subjects. Each notification is characterized by its *subject* and a *data part*. A component receives all the event notifications that belong to the subjects to which it has subscribed. Therefore, the subject defines a kind of virtual connector between objects of interest and recipients. The same approach based on the subject has been adopted by TIBCO for the development of TIB/Rendezvous [8].

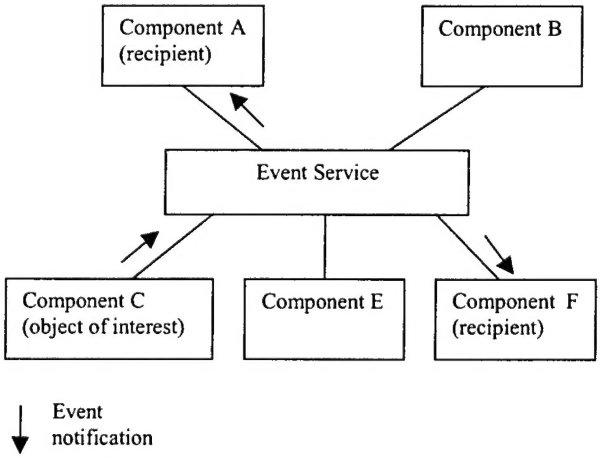


Figure 1. Architecture of an event-based system.

Event channels and subjects are simple mechanisms for connecting event receivers and objects of interest. However, they are not very flexible. If, for instance, an object of interest is interested in producing an event notification on a number of subjects or channels, it has to explicitly publish the notification on all of them.

An alternative approach is the one adopted by Elvin [[6]]. In Elvin, notifications are sets of named and typed data elements. A subscription is a declarative boolean expression over the components of event notifications. By issuing a subscription, a component can declare its interest in a number of notifications characterized by some common property. Notice that this property is established by the subscriber, and it is not hard-coded in any element of the notification (as in Smatsockets) or in a channel (as in the CORBA event service).

JEDI [1] provides a mechanism for event subscription having a similar expressive power. In JEDI a notification is defined by a name and by a number of parameters. For instance, `Alarm(PC1, HALTED)` is a notification whose name is `Alarm` and has two parameters whose values are `PC1` and `HALTED`. In JEDI, event receivers subscribe for *event patterns*, which are expressions over the name and parameters of a notification. So, for example, `Alarm*(_, _)` would match all the notifications whose name starts with `Alarm` and that have two parameters.

Yeast [2] is an *event-action system*. It observes event sequences and reacts to their occurrence according to some action specification. It is not an event-based infrastructure *per se*, since its event service triggers actions relevant to human beings rather than delivering notifications to other software components. However, it encapsulates interesting mechanisms for observing events. Differently from JEDI, in Yeast an event pattern can be composed of a number of *event descriptors* combined together using some logical and temporal operators. For instance, the event pattern "**`file`** `foo mtime` **`changed then in`** `10 minutes`" is matched 10 minutes after a change to file foo.

3. THE IMPACT OF MIDDLEWARE INFRASTRUCTURES ON APPLICATION ARCHITECTURES

We argue that the assumptions introduced at the implementation level by event-based infrastructures have an impact on the structure of the architectures implemented on top of them, and therefore define new event-based architectural sub-styles. In this section we briefly discuss about the architectural implications of three important aspects of middleware infrastructures:

- The mechanism that supports the selection of the recipients to be notified of the occurrence of an event (the *subscription mechanism*).
- The structure of notifications managed by the middleware infrastructure.
- The scalability properties of the middleware infrastructure.

3.1 Subscription mechanisms

The subscription mechanism influences the configuration of architectures and the interaction among components. As an example, suppose that we are building an application composed of three classes of components, `A`, `B`, and `C`.

Also, suppose that they interact through an event-based style and, in particular, that components of type A send two types of events, one that is supposed to be received by components of type *B* and the other that is received by components of type C.

In this case, if we use the CORBA event-based approach, the architecture of the system will reasonably contain two event channels, one connecting components of type A with components of type B, and the other connecting components of type A with components of type C. Since components of type B and components of type C will be connected to different connectors, they will receive separate sets of events. Components of type A will be in charge of selecting the proper event channel depending on the type of event it generates.

Conversely, by using Smartsockets, all the components will be connected to the same connector, the RT Server, and the dispatching of the events will not depend on the configuration of the architecture, but on the content of subscriptions.

3.2 Structure of notifications

The structure of notifications that are produced and consumed by components has an impact on the communication protocol established among them. For instance, if notifications contain minimal information related to an event, a recipient of an event would need to engage in a complex interaction with the object of interest in order to get additional information about the circumstances in which the event occurred. As an example, let us consider the case of a system for software deployment in which one component, *A*, is in charge of notifying the release of a new software product. Other components can subscribe to this event notification and, upon receiving it, can deploy the new software on the nodes where they are running.

In the case where we adopt a flat structure for notifications, the event notified by *A* can have the following appearance:

"Product A released on April 4th"

The components that receive this notification must parse it, extract the information about the software being released, and then engage some kind of communication with component A to know how to download and install the software.

The exploitation of an object-oriented notification structure provides more support to event recipients in the interpretation of the event semantics. In fact, if we adopt an object-oriented model, *A* can generate a notification containing the information on the released product plus a method to download and install the product. Upon receiving the notification, recipients can invoke this method to get the product installed without being aware of the location of the code or of the downloading and installation procedure.

3.3 Scalability properties

The internal architecture of the event service significantly influences the performance of the architectures built on top of it. Intuitively, it has to *scale* to accommodate a growing number and distribution of components. If we assume that the event service is implemented as a centralized element, it can rapidly become a critical bottleneck as the number of components it has to serve grows. This scenario becomes even worse when components are distributed over a wide-area network.

To solve this problem, in JEDI the event service itself is built as a set of distributed components, the *event servers*, organized in a hierarchy. Each event server manages the communication among a set of components geographically located in the same "neighborhood" and also connects them to other remote components by forwarding messages to and from other servers. In particular, subscriptions are stored and forwarded upward in the hierarchy until they reach the root server while notifications are sent to all the local subscribers, to all the lower-level servers that have forwarded a corresponding subscription, and then upward in the hierarchy.

In JEDI the distribution of the event service is hidden to components and does not have an impact on the functional behavior of the system. In particular, the notification delivery functionality of the event service is guaranteed regardless of whether the objects of interest and recipients are located closely to each other. This requires a consistent amount of information to be exchanged among event servers.

In the case in which recipients and objects of interest of the same event are confined to the neighborhood managed by a single event server, the performance of the whole system could be even worse than in the centralized approach, since messages would be unnecessarily propagated to the top of the hierarchy. Thus, other propagation mechanisms for subscriptions and publishing of notifications need to be identified and evaluated. Also, the assumption that the event servers are organized hierarchically should be assessed against other alternative topologies.

4. CONCLUSION

We argue that the event-based style presents interesting characteristics for the development of distributed, highly-decoupled systems. Several infrastructures that support the development of event-based applications have been developed. Each of them makes specific assumptions on the structure of notifications, on the mechanism that allows components to declare their interest in some event, and on the way scalability of architectures is supported. All these assumptions have an impact on the final architecture of the applications that are developed on top of these infrastructures. Conversely, considerations concerning the architectural structure of applications influence the choice of the underlying event-based infrastructure. In this paper

we have briefly discussed these issues, pointing out the advantages and drawbacks of each approach.

The evaluation we are currently carrying out results in a more general consideration that we have begun to analyze in more detail: *middleware infrastructures (not necessarily event-based) implicitly define architectural (sub)styles*. The knowledge of these styles can be profitably used when the architecture of an application is defined. It is therefore desirable to explicitly define them in terms of architectural elements, so that they can provide guidelines to application developers and can support the transition of an architecture into an implementation on top of a selected infrastructure.

5. ACKNOWLEDGEMENTS

We thank Gianpaolo Cugola and Alfonso Fuggetta from Politecnico di Milano for their important contribution to the accomplishment of the work described in this paper.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-98-1-0061. This work was also supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems", In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[2] B. Krishnamurthy and D. S. Rosenblum, "Yeast: A General Purpose Event-Action System", *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, October 1995.

[3] Object Management Group, "CORBAservices: Common Object Services Specification", December 1997.

[4] D.S. Rosenblum and A.L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification", In *Proceedings of the 6th European Software Engineering Conference* (Joint with SIGSOFT '97, Foundations of Software Engineering), Zurich, Switzerland, September 1997.

[5] M. Shaw and P. Clements, "Toward Boxology: Preliminary Classification of Architectural Styles", In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco (CA), USA, October 1996.

[6] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quencing", In *Proceedings of AUUG97*, September 1997.

[7] Talarian Corporation, "Mission Critical Interprocess Communications - an Introduction to Smartsockets", white paper.

[8] TIBCO Corporation, "TIB/Rendezvous", white paper. http://www.rv.tibco.com/rvwhitepaper.html.